EXHIBIT 31

SUPPORTING UBIQUITOUS COMPUTING WITH STATELESS CONSOLES AND COMPUTATION CACHES

A DISSERTATION
SUBMITTED TO THE COMPUTER SCIENCE DEPARTMENT
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Brian Keith Schmidt

August 2000

Abstract

Rapid advances in processing power, memory and disk capacity, and network bandwidth are leading us to an era in which we will have access to virtually unlimited resources at our fingertips. We are moving away from a hardware-centric view of computing to a more service-oriented perspective, i.e. users simply want to accomplish their tasks on their data, and they do not care about the machine specifics and management details. The future computing infrastructure will provide ubiquitous access to active computations floating on a sea of globally distributed resources. When executing in this context, our computing environments must become hassle-free and much easier to manage. This thesis describes two core concepts to help enable this transition.

First, we propose decoupling displays from their servers, i.e. using network-attached frame buffers to access active computations from any location. These simple consoles are cheap, fixed-function appliances like telephone handsets. They are stateless, low-level access devices that require no administration, allow users to transparently migrate between terminals, and can isolate users from desktop failures. We developed a methodology for evaluating the interactive performance of computer systems, and we use it to demonstrate that an architecture based on this concept is feasible with modern technology and can provide an interactive experience that is indistinguishable from the traditional desktop, even for highly interactive video games. This establishes the viability of providing ubiquitous access to active computations that reside within the network.

The next step is to manage the computing resources at the back end, and we place several requirements on this remote computational service. Users must have privacy, security, and isolation from each other, and they should be free to roam throughout the system and to customize their environments. Active sessions will reside in the system, even when users

are detached, and so the system must scale to support an unlimited number of users while requiring resources in proportion to the number of active users. The system must support user mobility and dynamic load balancing for efficiency, which means sessions must be machine-independent and portable. Reliability, availability, maintainability, and extensibility are major concerns, e.g. for automatic fail-overs, on-line maintenance and upgrades, application-independent checkpoint/restore, etc. To meet these goals, we restructured the operating system using the concept of *compute capsules.* Capsules provide a private, portable, persistent, customizable computing environment with active processes, i.e. they are self-contained, running computations. The CPU/OS merely serves as a cache to hold active computations. Capsules can be dynamically bound to any machine to support user mobility or load balancing, stored off-line to free resources or survive failures, assigned resources to provide performance isolation, managed to enforce security policies, etc. We demonstrate that it is feasible to extend COTS operating systems to support this new abstraction, and we show how capsules can help meet the demands of the future computing infrastructure by supporting persistent computations on a globally distributed collection of anonymous computing resources.

To my family,
for their love and support.

Acknowledgments

I would like to thank my advisor, Monica Lam, for her enthusiastic support of this research. I feel fortunate to have had the opportunity to work with someone of her intellectual caliber, and I have benefited greatly from her guidance and direction.

I am grateful to John Hennessy and Mendel Rosenblum for sitting on my Reading Committee and offering helpful comments on previous drafts of this dissertation. Dawson Engler graciously gave his time to serve on my Orals Committee, and John Gill kindly offered to act as chairman for my Oral Defense.

I am extremely indebted to my colleagues at Sun Microsystems: Duane Northcutt, Jim Hanko, Jerry Wall, and Alan Ruberg. Duane has been a constant supporter of my research, and his personal and intellectual involvement have been invaluable over the years. Jim, Jerry, and Alan have offered much of their knowledge, wisdom, and time; and it has been a true honor to work with them. Several other folks at Sun have made my time there an extremely rewarding experience, including Lawrence Butcher, George Salem, Marc Schneider, Yuanbi Su, and Chuck Yoo.

I was fortunate enough to have wonderful office mates during my time at Stanford. Jason Nieh and I shared not only an office but a good friendship, and we worked closely on many projects, most importantly the development and testing of 3D graphics benchmarks. I also shared an office with Shih-Wei Liao, Amy Lim, and Jeff Oplinger, who were always ready with good conversation, good humor, and good company. I would also like to acknowledge the other members of the SUIF group, particularly Dave Heine for all his help when I was pretending to be a compiler person and Costa Sapuntzakis for many insightful discussions on this work.

Most importantly, I would like to thank my family for their steadfast love and unwavering support throughout my time at Stanford. It has been a long, arduous, and sometimes painful journey, but my wife, Melisa, has always been there to lift me up and carry me. She gave me the freedom to pursue my goals, even though it meant so much lost time for ourselves. She brightened my days with cheerful visits and listened when I droned on incessantly regarding the minutiae of my work. I love her beyond all things, and I am truly grateful for all that she has given me. My daughters, Madison and Kasi, are my most precious treasures. They have filled my life with happiness and pride — more than they could ever understand. They will always have my unconditional and unbounded love.

Finally, this work was supported in part by financial assistance from a National Science Foundation Graduate Research Fellowship, a National Science Foundation Young Investigator Award, and Sun Microsystems, Inc.

Table of Contents

List of Tables

List of Figures

1 Introduction

As machines and networks continue to improve in both price and performance, users will enjoy fast, interactive access to a globally distributed set of virtually unlimited computing resources. In the traditional computing model, users had access to powerful desktop computers, and a great deal of personal computing was performed by local hardware. Today, we have already begun to see a trend toward moving personal computing off local desktop resources and onto remote servers. The worldwide web is a good example of the trend in this direction, as computing is increasingly being performed on a user's behalf by anonymous machines in distant corners of the internet.

As a result, our computational environment is undergoing a radical change. We are seeing a shift from the machine-centric view of computing to a more service-oriented perspective. Currently, users must concern themselves with many unnecessary details, such as the names, addresses, architectures, operating systems, and software configurations of the machines they use. In addition, users are burdened with the complex tasks of system administration, which has become a major factor in the total cost of owning a computer. However, users simply wish to accomplish their tasks on their data without such worries. Thus, resources are being increasingly pooled and accessed remotely in order to centralize system administration and management. In this setting, the network can be viewed as a repository of computing services that are made available to users in a manner which is free of the hassles to which they are normally accustomed. Just as the web can provide specific application services today (such as e-mail), our general computing environment will soon be available from anywhere in the network. This thesis explores some of the critical issues necessary to begin moving in that direction.

1.1 The Future Computing Infrastructure

Our goal is to create an infrastructure in which active computing sessions are persistent and reside within the network. These sessions will contain the standard set of applications users run today on the desktop. Users will simply attach to their sessions from any network portal to access ongoing computations as well as personal data, and the active computing sessions will float through the network, binding to different machines in response to user movement, dynamic load balancing, on-line maintenance, system failures, etc. The host computers within the network will be professionally administered and shared to reduce management costs for the end users.

The future computing environment will consist of simple display consoles attached to the network in every conceivable location, as well as wireless consoles for mobile users – much like telephones today. These consoles will provide high performance access to active computing sessions, while being fully interchangeable, requiring zero administration, and isolating users from desktop failures. The hosts within the system will serve processor cycles and memory to active sessions, thereby acting as caches for running computations. System administration then becomes a matter of managing a cache consistency protocol.

Consider the properties of a data cache. Information is cached near its computation for performance, and data are evicted from the cache when its capacity is reached. Data can be written back to more stable memory for persistence, and weak consistency protocols can be used to support disconnected operation. Finally, this functionality is provided transparently. The same can be said for caching computations. They can be cached near a user for performance (i.e. mobility), evicted from a host which has reached capacity in some resource (i.e. dynamic load balancing), written to disk for persistence (i.e. fault tolerance and scalability of resource consumption), and cached on a stand-alone machine (i.e. disconnected use). Our future computing infrastructure will support such caching of active computing environments in a manner which is transparent to both users and programs so that the system can be easily deployed without requiring changes to the existing base of applications.

1.2 A New System Architecture

As a first step toward developing a computing infrastructure that matches the vision described above, we have devised a remote computational service architecture based on the notions of stateless display consoles and cacheable compute sessions. This new system architecture enables us to explore some of the initial, critical research issues involved in the development of our future computing environment. In addition, we are able to achieve concrete results today with current technology. The architecture is depicted in Figure 1-1, which shows two clusters of resources attached to the global network. The clusters are located on the periphery of the global interconnect and may be geographically separated. The resources within a cluster include a set of display consoles, a collection of session server machines, and a group of back-end servers.

The display consoles will replace standard desktop computers. For example, they may be deployed in a corporate workgroup setting or across a neighborhood of home users. They are thin-client devices that are connected to the session servers via a dedicated display network, which carries display traffic with guaranteed quality of service. The consoles will embody the ideal thin-client terminal: low cost, zero administration and management, no standard performance upgrades every few years, fully interchangeable units, seamless user mobility, immunity to desktop failures, no unwanted noise, etc. Yet, they will support the standard desktop applications available today. They are completely hassle-free devices that provide access to the session servers, which are responsible for executing the actual applications that are normally run on desktop computers.

Active sessions are free to migrate between session servers of the same architecture, as well as to disk. In addition, the session servers provide access to high-performance back-end servers for running compute-intensive jobs or maintaining large databases. To support disconnected operation, an active session may migrate to a stand-alone session server, such as a laptop computer, which performs the functions of the display console as well as any back-end server tasks it can support, e.g. large jobs may run locally, while database transactions are queued until the server is re-connected to the network.

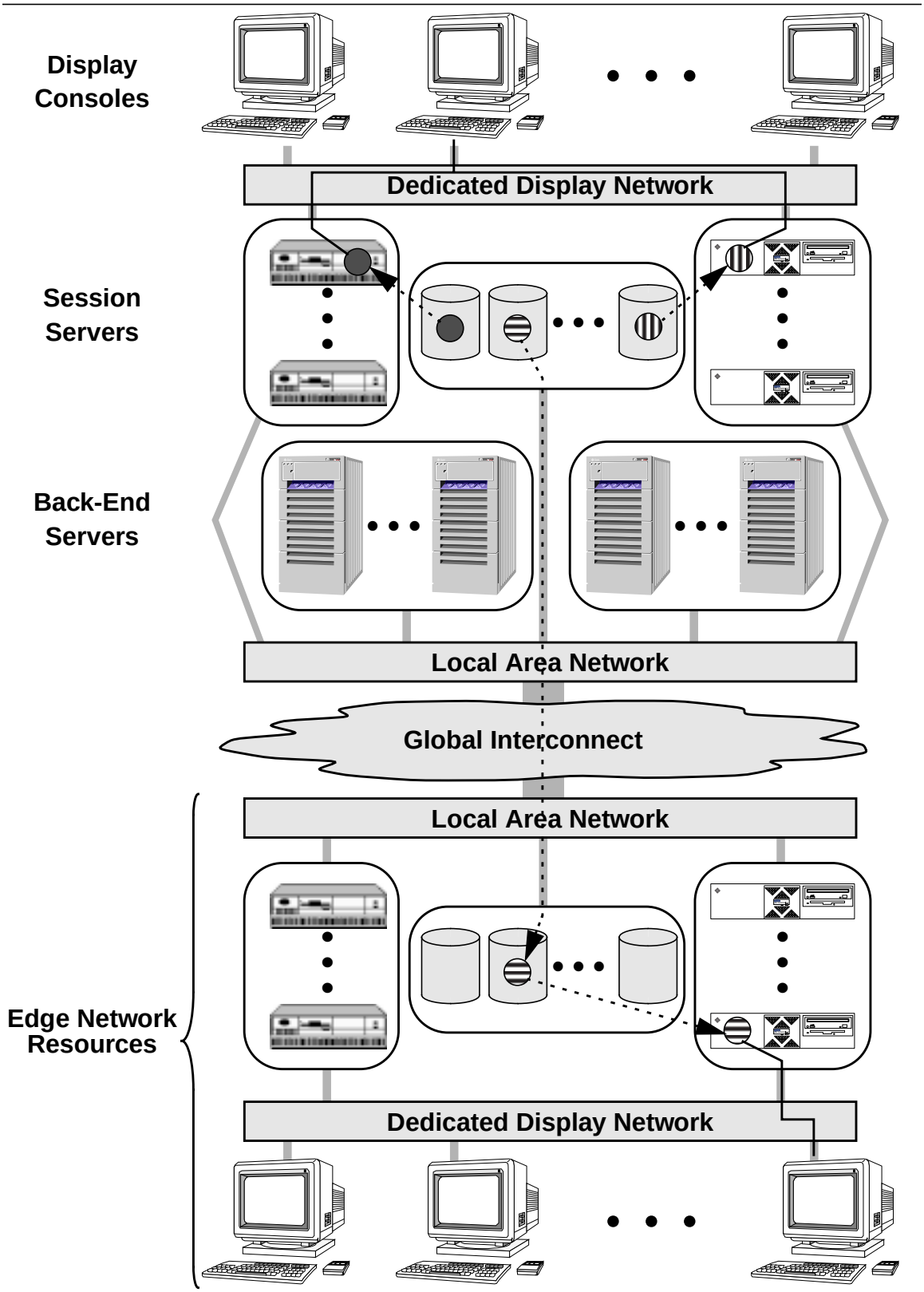


Figure 1-1 Future remote computational service architecture based on stateless display consoles and cacheable compute sessions.

1.2.1 Example Usage Scenario

To give a sense of the types of activities we would like the system to support, consider the usage scenario depicted in Figure 1-1. A user walks up to a display console and presents identification to the system, perhaps via a smart card or biometric signature. The system will authenticate the user and search for any active computing sessions the user may own. In this case, the system finds two sessions, perhaps one for personal use and one for work-related tasks. Because the user has been idle for a while, the sessions are currently on disk in suspended animation, i.e. the active computations were frozen during mid operation (after a period of user inactivity), and the complete state was moved into secondary storage.

One session (indicated by the circle with horizontal stripes) consists of computations that execute on a single type of architecture, and the other session (indicated by the solid circle and the circle with vertical stripes) contains computations that execute on machines of two different architectures (indicated by the two styles of session servers). The user may decide to attach to the second, multi-architecture session. The system responds by retrieving the session from disk and reviving it on machines of the appropriate architecture, perhaps based on a load balancing facility. Once the session is restored to the running state, the display streams from the session servers are merged and directed to the user's console, and normal operation can resume. In this way, the user sees only the display interfaces to the active computations and is unaware of which specific machine or architecture is responsible for executing a particular element of the session.

Now, suppose that the user's console suffers a catastrophic failure, perhaps due to an errant beverage container. The session servers are unaffected, and the session remains active and intact; only the display stream has been severed. The user can simply move to a new console (or replace the failed unit), and the display stream will be seamlessly directed to the new location. There is no disruption of service, and the user can resume working as if nothing had happened. A failure of the desktop unit will have the same impact that the failure of a monitor has today. In fact, desktop consoles are analogous to standard monitors in terms of the support they require, i.e. there is nothing to be administered or customized on the desktop, and consoles are completely interchangeable.

To reduce costs and user frustration, the session servers are professionally managed in a centralized location. We expect a very large user pool compared to the number of session servers, e.g. on the order of tens to hundreds of users per server, only some of whom will be active at any given time. This enables the system to take advantage of economies of scale and dramatically reduce administration costs. In addition, the burden of system maintenance has shifted from the user's desktop to a remote service provider that is responsible for ensuring a consistent, working, and reliable computing environment. In this setting, computing is hassle-free for the user and easier for the system administrators to manage.

After the system has been in operation for a while, one of the session servers may exceed capacity in some resource for an extended period of time. For example, swap space may be full, or the processor may be saturated. Similarly, a session server may require a software upgrade, which necessitates a temporary shutdown of the machine. Rather than have the user suffer poor performance or complete loss of the active session, the system supports dynamic load balancing and on-line maintenance by migrating the session off its current hosts and onto other available machines.

When the user completes working and disengages from the display console, the system responds by terminating the display stream and recording the complete state of the session to stable storage. The session will now survive server failures, and it can be terminated to free resources. When the user accesses the system again, the session can be revived from disk and returned to the exact state at the time it was stored.

Suppose the user has travelled to a distant location and wishes to access an active session that is currently stored on the home cluster of session servers (indicated by the circle with horizontal stripes). The system locates the suspended session on the remote home site, moves it to a local disk, revives it on an appropriate session server, and directs the display stream to the user's current console. The user may travel across the globe on business or across town between home and work. In each case, the new location may be served by a completely different set of resources, and the system will respond by migrating the active session to local servers and directing the display stream to the user's local console.

1.2.2 System Requirements

To better codify the properties of our proposed architecture, we distill the requirements a system must meet in order support this new computing model.

(1) *Easy Access.* Desktop terminals must be low-cost devices that offer high performance access to active computations. At the same time, they must require no administration so that distributed maintenance costs are reduced.

(2) *Anonymous Resources.* The system will consist of a wide variety of machines with different architectures, and users should not need to be aware of such details. Thus, hosts in the system must become anonymous computing resources, and sessions should simply run on any machine of the appropriate architecture. The terminals must also be nameless and architecture-neutral so that they can merge the displays from different machines.

(3) *Persistent Computations.* In this new model, computations are long-lived and will persist in the network even when users are not actively interacting with them. For example, there will no longer be a need for users to log out of the system, and sessions may reside on disk for indefinite periods of time.

(4) *Easy Deployment.* To migrate to this new computing infrastructure, it must be easy to deploy. It should require only commodity technology available today, and it should support standard desktop applications without requiring changes. Applications should not need to be modified in any way.

(5) *Shared Resources.* Resources should be shared as much as possible to amortize not only hardware costs but also the high cost of administration and management. At the same time, users would like to maintain levels of privacy and isolation comparable to the traditional dedicated computing model, which ensures users cannot view, manipulate, or indirectly affect the environments of others. Similarly, while professional administration reduces costs, users require the ability to customize their environments freely.

(6) *Transparent Mobility.* We live in a mobile society, and users should be able to access their active computing environments from any terminal worldwide. This may require the re-direction of a display stream or the migration of an active session. In addition, system management demands may require the movement of active sessions, and users should not be aware that such migration has occurred.

(7) *Reliability.* Desktop failures should have absolutely no effect on users or their sessions, other than the overhead of finding another terminal to use. In addition, active sessions should have the ability to survive server failures, thereby improving the reliability and availability of the overall system.

(8) *Scalability.* Idle sessions still occupy valuable resources (such as kernel memory, swap storage, process slots, etc.), and the system should archive them to free resources when needed. This improves scalability of the system by requiring it to scale with the number of active sessions, not the total number of sessions, which may be greater by orders of magnitude. In addition, support services (e.g. migrating active sessions) must scale to internet dimensions, which means that the hosts in the system must be loosely coupled and largely independent.

1.2.3 Limitations of Current Systems

To meet the above requirements, our computing environments must become portable and persistent while maintaining privacy, isolation, and customization capabilities within a shared environment. In essence, we would like to combine the best features of dedicated personal computing and time-sharing without the attendant drawbacks. However, these objectives cannot be met with current systems designs, as demonstrated by the following examples.

- Traditional desktop devices with traditional software systems do not provide zero administration, transparent mobility (of users or computations), or mechanisms to survive failures with no ill effects.

- Resource names are often tied to a specific instance of an operating system, which inhibits the portability of the computing session.
- Critical state of an active computation is inaccessible within the kernel, and so the session is locked to a specific machine, which binds its existence to the transient lifetime of the underlying operating system.
- Within a shared operating environment, there is no personal privacy or isolation, as users may freely inspect and sometimes manipulate the processes of others.
- Shared systems are usually centrally managed to reduce administration costs, but this also eliminates the opportunity for customization, as users must rely on the system administrator to alter their environment.

1.3 Focus of this Research

The purpose of this research is to develop the infrastructure upon which our desired system could be constructed. In particular, we focus on two aspects of the problem: how to decouple the human interface from servers to achieve the features of the display consoles described above, and how to support remote computational services on the session servers with persistence, mobility, customization, etc.

1.3.1 Decoupling the Human Interface

By decoupling the human interface elements (mouse, keyboard, display) from the computing resources, we can create a desktop unit with all the properties we described above. In particular, we propose attaching the frame buffer, mouse, keyboard, and audio devices directly to the dedicated display network (see Figure 1-1) for accessing computing resources on the session servers. The desktop units can then be simple, fixed-function devices that are completely stateless and architecture-neutral. We call a system based on this principle SLIM (Stateless Low-level Interface Machine), and Sun Microsystems has implemented a SLIM system as the Sun Ray product line. In this section, we describe the motivation for and present an overview of the SLIM remote display architecture.

1.3.1.1 Background

Since the mid 1980s, computing environments have moved from large mainframe, time-sharing systems to distributed networks of desktop machines. This trend was motivated by the need to provide everyone with a bit-mapped display, and it was made possible by the widespread availability of high-performance workstations. However, the desktop computing model is not without its problems, many of which were raised by the original UNIX designers [42]:

> "Because each workstation has private data, each must be administered separately; maintenance is difficult to centralize. The machines are replaced every couple of years to take advantage of technological improvements, rendering the hardware obsolete often before it has been paid for. Most telling, a workstation is a large self-contained system, not specialised to any particular task, too slow and I/O-bound for fast compilation, too expensive to be used just to run a window system. For our purposes, primarily software development, it seemed that an approach based on distributed specialization rather than compromise could better address issues of cost-effectiveness, maintenance, performance, reliability, and security."

Although many of the issues outlined above are no longer applicable today (particularly since higher-performance personal computers are readily available at low cost), it has been well-established that system administration and maintenance costs continue to be dominant factors in the total cost of ownership, especially for networks of PCs. In addition, there are many large computational tasks that require resources (multiprocessors and gigabytes of memory) that cannot be afforded on every desktop.

A great deal of research went into harvesting the unused cycles on the desktop machines of engineers [2]. However, the return to time-sharing a centralized pool of hardware resources is the most effective and direct solution to this problem. Thus, we would like to use network-attached access devices to connect to active computing sessions on remote servers, i.e. decouple the display and human interaction devices from the computer itself. This enables us to remove high-maintenance PCs and workstations from the desktop and

replace them with simpler devices that require no administration and no management while maintaining high performance at low cost.

1.3.1.2 Design Requirements

To devise a system based on the principle described above, we must set forth the design requirements for decoupling the human interface from servers. Thus, we outline the six primary goals for such an architecture below.

(1) *Zero Administration.* As mentioned earlier, system administration has become the dominant factor in the total cost of owning a computer. The desktop devices should require absolutely no administration or management by the end user. For example, the user should not need to perform hardware or software upgrades, configure the device for operation, resolve network issues, etc.

(2) *Minimal Resources*. The desktop devices should contain as few resources as possible in order to minimize costs and reduce complexity.

(3) *High Interactive Performance*. Although we want to minimize the resources on the desktop, we cannot sacrifice performance. Because the desktop unit merely provides access to remote resources, its computational throughput is not particularly relevant. Instead, we wish to maintain high interactive performance, as perceived by the end user.

(4) *Fault Tolerance*. Users should be completely isolated from failures at the desktop. If a terminal fails, the user should be able to replace it and resume work as if there had been no disruption in service, and active computations should not be affected by the failure. (Server failures are considered below when we discuss support for a remote computational service.)

(5) *User Mobility.* Because we wish to provide ubiquitous access to active, network-resident computing environments, users should be able to attach to their sessions from any terminal, and they should be free to roam between terminals in a seamless manner.

(6) *Easy Deployment*. To be widely adopted, decoupling displays from their servers must not be disruptive to current practice. Applications should continue to run unchanged, and the terminals must be independent of any architecture, operating system, display API, or application.

1.3.1.3 Thin-Client Computing Models

Thin-client computing is one means of achieving the goal of using a network-attached device to access remote computational services. The guiding principle of this architectural model is to pull as many computing resources off the desktop as possible and centralize them in a shared machine room, which is accessed remotely over the network. This is much like the time-sharing model of the 1970s in which dumb character terminals were used to interact with mainframe systems across serial lines. However, to support modern graphical and multimedia applications, thin-client devices expend computing resources on the desktop to improve interactive performance and reduce bandwidth requirements. There have been numerous projects that focus on thin-client strategies, and we discuss a few of them below.

The Plan 9 project completely redesigned the computing system with an emphasis on supporting a seamless distributed model [42]. Although it is intended to run only programs with low resource demands, the terminal at the desktop in Plan 9 is still a general-purpose computer running a complete virtual memory operating system, and thus it is subject to the same administration and maintenance demands of standard computer systems. Simply reducing the desktop computing capabilities is insufficient to achieve the goals of thin-client computing. A fundamental change in the computing architecture is necessary.

Thus, more recent thin-client system architectures have split application-level functionality between the desktop units and the server machines. Examples include X Terminals, Windows-Based Terminals, JavaStations, and other Network Computers. In these systems the desktop unit executes only the graphical user interface of applications, thereby enabling more sharing of computing and memory resources on the server at the expense of added network traffic in the form of display protocols, such as X [39], ICA [6], and RDP [36]. These protocols are highly optimized for specific software APIs to reduce their bandwidth

requirements. Thus, they typically require a nontrivial amount of processing resources and memory to maintain environment state and application data, such as font libraries or Java applets. In addition, users are still subjected to some degree of system administration, such as network management and hardware/software upgrades.

Finally, the thinnest possible clients are dumb terminals that only know how to display raw pixels. The MaxStation from MaxSpeed Corporation is refreshed via 64Mbps dedicated connections, which can only support a resolution of 1024x768 with 8-bit pixels [35]. However, a standard 24-bit, 1280x1024 pixel display with a 76Hz refresh rate would require roughly 2.23Gbps of bandwidth. The VNC viewer, on the other hand, allows access to a user's desktop environment from any network connection by having the viewer periodically request the current state of the frame buffer [47], which is maintained at the server. While this design allows the system to scale to various network bandwidth levels, its interactive performance (even on a low-latency, high-bandwidth network) is noticeably inferior to that of a desktop workstation.

1.3.1.4 Thin-Client Deficiencies

The approaches described above do not completely satisfy the requirements listed in Section 1.3.1.2. They are often not very lean with respect to resources, sometimes entailing complete computer systems; and their cost is often not substantially less than modern PCs. Although they reduce management overhead, they are not true zero-administration devices. They require hardware upgrades for performance, substantial configuration effort, network management, etc. Because nearly all thin-client approaches maintain state on the client device, surviving desktop failures requires enhancements to these designs. State must be carefully duplicated at the server, and consistency must be maintained. In addition, dedicated connections and associated state are maintained between applications on the server and display routines on the client, which locks a user to a particular terminal. Severing the connection to one terminal and re-targeting it to a new terminal is not supported, and it would be difficult to achieve transparently because connection-oriented network protocols are used. Finally, each terminal employs a domain-specific protocol that supports a specific class of applications. This limits the scenarios in which they can be

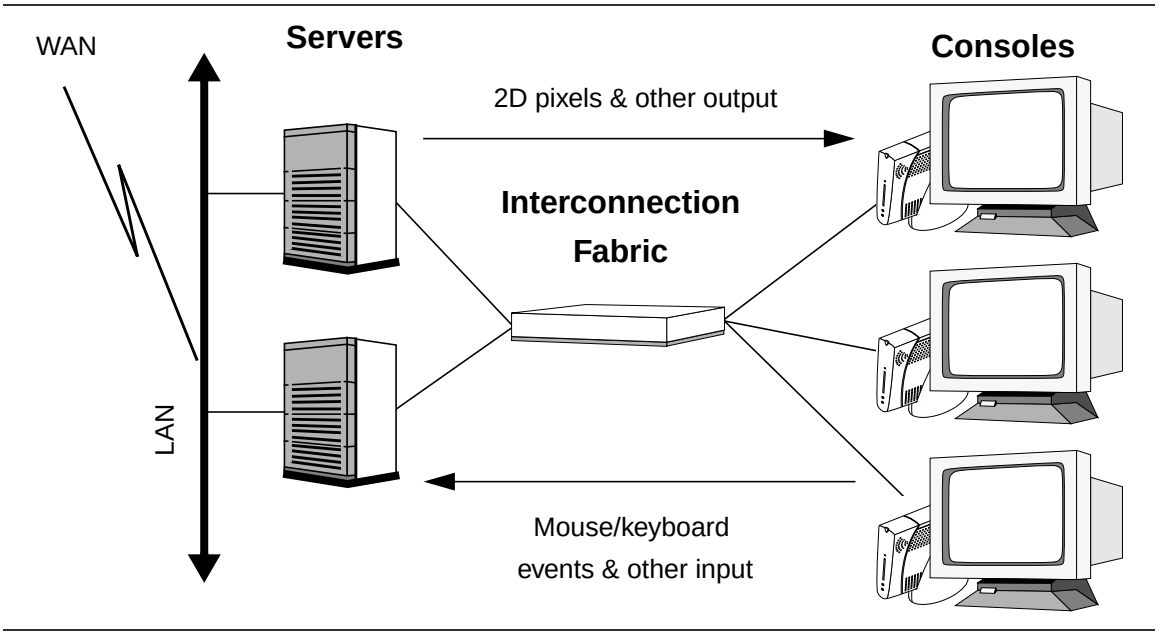


Figure 1-2 Major components of the SLIM architecture.

deployed, and it requires a proxy to convert the protocol for systems that do not support it directly.

1.3.1.5 SLIM: Taking Thin-Clients to the Limit

To create an architecture that meets all the requirements outlined above, we explore means for extending the thin-client model to alleviate its deficiencies. The premise of our approach is that commodity networks are fast enough to use a low-level protocol to remotely serve graphical displays of common, GUI-based applications without any noticeable performance degradation. This leads us to take the notion of thin clients to the limit by removing all state and computation from the desktop and designing a low-level hardware- and software-independent protocol to connect all user-accessible devices to the computational resources of a system over a low-cost commodity network. As mentioned above, we refer to such thin-client architectures as SLIM systems, and we illustrate the major components in Figure 1-2.

SLIM consoles communicate with the SLIM sessions servers over a dedicated interconnection fabric. The servers execute all software (including the operating system)

on behalf of the user and transmit all interface device output in raw format (e.g. display pixel data) to the consoles. The consoles, in turn, transmit all input from human interface devices (e.g. keyboard and mouse) across the interconnection fabric to the servers for event processing. The goal is to reduce the console to the minimum set of resources by exchanging the elimination of resources from the desktop for higher bandwidth requirements in the form of a raw I/O protocol. The expectation is that modern network technology will make this trade-off cost effective.

As an extreme design with the thinnest possible terminals, SLIM maximizes the advantages of thin-client architectures: resource sharing, centralized administration, and inexpensive desktop units to minimize cost per seat. In addition, the SLIM architecture has the following distinctive characteristics:

- *Statelessness*. Only transient, cached state (such as frame buffer content) is permitted in SLIM consoles; the servers maintain the true state at all times. Thus, the user is not affected by desktop failures and may move freely between terminals. In our Sun Ray implementation, users can simply present a smart identification card at any desktop, and the screen is returned to the exact state at which it was left.

- *Low-level interface.* The low-level SLIM protocol is designed to move raw I/O data between the servers and consoles. Thus, a SLIM console is merely an I/O multiplexor connected to a network. The protocol can be implemented by simply redirecting server I/O to SLIM consoles over the network at the device driver level. Thus, applications on the server require no modification. SLIM consoles are not tied to a particular display API (e.g. X, Win32, Java AWT), and so applications running on different system architectures with different APIs can be easily integrated onto a single desktop.

- *A fixed-function appliance*. As it merely implements a fixed, low-level protocol, a SLIM console bears more similarity to a consumer-electronics device than a computer. There is no software to administer or upgrade; its simplicity makes it

amenable to a low-cost, single-chip implementation; and finally, a SLIM console requires no fan and generates no unwanted noise.

These characteristics enable the SLIM architecture to satisfy our six design goals listed in Section 1.3.1.2. Because SLIM consoles are stateless, fixed-function devices, there is nothing for users to configure, maintain, or administer. As fixed-function devices, they are also amenable to low-cost, single-chip implementations. Users are completely immune to desktop failures, as there is no state in the console to be lost. The low-level interface protocol enables the users to easily move between consoles, and the interface stream can be re-directed to the appropriate desktop unit. In addition, because the interface protocol is so low-level, it can be easily deployed without requiring modifications to applications or domain-specific support. Still, there is an open question with respect to the performance of such a system. Will users find their interactive experience with a SLIM console to be satisfactory? A major part of our research is to explore the answer to this question.

1.3.1.6 Interactive Performance Evaluation

Although characterizing the performance of traditional computer systems is fairly well understood, the throughput-based benchmarks used in such evaluations are inappropriate for a SLIM system. Because SLIM consoles do not execute any software and merely serve as network portals for human interface devices, the user's interactive experience is a more critical measure of performance than raw processing power. In addition, the separation of the console from the session server via an interconnection fabric must be taken into account. Thus, new metrics for evaluating interactive performance are needed to assess the feasibility of a SLIM system. Of course, the standard benchmark techniques can be applied to the SLIM session servers, which run all the software, but we are concerned with the facets of the architecture that distinguish SLIM from the traditional desktop model.

Thus, we have developed a new experimental methodology for evaluating the interactive performance of a remote display architecture, focusing on the effect the decoupled interface device has on response time. Using common, GUI-based applications as benchmarks and live users in a real-world setting with Sun Ray appliances, we demonstrate that (1) a user's interactive experience with a SLIM console is indistinguishable from sitting at the console

local to the server, (2) bandwidth requirements are quite modest and competitive with higher-level display protocols such as X, and (3) minimal resources are required in the desktop unit. Next, using a load generator to play back recorded user sessions, we demonstrate that substantial sharing is possible on both the server processor and network resources before quality of service degrades to an unusable level. Finally, we stressed the system with multimedia and video game applications to demonstrate that even the resource-poor Sun Ray terminal can provide a high-fidelity experience for such demanding applications.

These results demonstrate that the SLIM approach is a more cost-effective means of supporting a workgroup environment than either networks of personal computers or thin-client solutions, such as X Terminals or Windows-based Terminals. All three approaches provide equivalent levels of performance, and all three approaches require the same network infrastructure (in the form of the SLIM interconnection fabric or a standard local area network). However, when compared to a network of personal computers, a SLIM system consolidates computing resources to reduce not only hardware costs but also (and more importantly) administration costs. Thin-client systems enjoy a similar benefit, but there is still additional cost associated with purchasing and maintaining the desktop units. When a SLIM console is implemented as a single ASIC, it can be embedded within a monitor so that there is effectively no cost associated with the desktop unit.

1.3.2 Supporting Remote Computational Service

Decoupling the human interface from the compute resources provides a means of accessing active computing sessions from any network portal. However, that is only the first step toward achieving the type of system architecture we outlined above. We must also provide server support so that active computing sessions in a shared infrastructure can migrate between machines, survive failures, support customization, etc. Thus, we have developed a new operating system abstraction called a *compute capsule*, which fully encapsulates an active computing environment and enables the system to manage it directly. In this section, we present some background and an overview of the compute capsule abstraction, as well as some beneficial features of capsules and sample usage scenarios.

1.3.2.1 Background

Resources are being increasingly pooled and accessed remotely in order to centralize system administration and management. We believe the future computing infrastructure will provide ubiquitous and hassle-free access to active computing sessions executing on an essentially unlimited set of anonymous computing resources. Within this context, issues of scalability, reliability, maintainability, user mobility, and efficiency are more critical than the traditional focus on system performance.

Our currently active computing sessions will float through the network, binding to different machines in response to user mobility, dynamic load balancing, on-line maintenance, system failures, etc. Users will be able to concentrate solely on their data and applications, as machine specifics (such as names, locations, and architectures) and management details (such as system administration, maintenance, and upgrades) will be handled automatically. To reach this goal, our computing environments must become portable and persistent while maintaining privacy, isolation, and customizability within a shared environment. In essence, we would like to combine the advantages of personal computing with the benefits of centrally managed resources. Simultaneously satisfying these goals is not possible with current systems, which requires us to re-think the organization of operating systems. Thus, we propose extending commodity operating systems with the new compute capsule abstraction.

1.3.2.2 Compute Capsule Overview

A compute capsule is a private, portable, persistent computing environment that contains active processes. Capsules virtualize the application interface to the operating system and re-partition state ownership so that all host-dependent information is moved out of the kernel and into capsules. Thus, they are entirely self-contained and can be suspended in secondary storage, arbitrarily bound to different machines and different operating systems, and transparently resumed. Capsules include a private namespace and personal view of the underlying file system, and they may be assigned resources and subjected to security policies.

In our proposed system architecture, active computations are encapsulated in compute capsules and may be cached on any of the processors in a distributed network. Capsules can be moved for on-line maintenance, dynamic load balancing, and proximity to mobile users. They can be placed in stable storage to free scarce resources or to survive system failures. The private capsule namespace provides isolation between users and customizability in a shared environment. By assigning resources and privileges to the capsules, the system can support guaranteed performance and different levels of trust.

Thus, the processor and operating system can be viewed as a cache for active computations, and capsules represent the cacheable entities. System management becomes a simple cache consistency protocol. Computations can be cached near users for performance (e.g. dynamic load balancing and user mobility). Cached computations can be evicted when capacity is reached (e.g. over-utilization of the processor or kernel resources). Write back policies can ensure cached computations persist beyond the lifetime of the underlying hardware or operating system (e.g. fault tolerance, on-line maintenance), and caching is fully transparent to the computation.

A traditional operating system provides resource management functionality as well as the user computing environment, but we separate these components in our new system architecture (see Figure 1-3). A user's personal computing environment resides fully within the capsule, while the underlying operating system provides per-processor resource management functionality. In addition, a distributed network services layer coordinates activity between machines, e.g. for user mobility. In this dissertation we concentrate on the design, implementation, evaluation, and sample uses of capsules, while the service layer is outside the scope of our current work.

In contrast, distributed operating systems attempt to present a single system image to make a cluster of machines appear as one machine to the user. This approach has the benefit of easily moving active processes between machines for load balancing, availability, and fault tolerance. However, there has not been a truly scalable system built with this approach to date. The single system image approach restricts the behavior of migratory processes, and

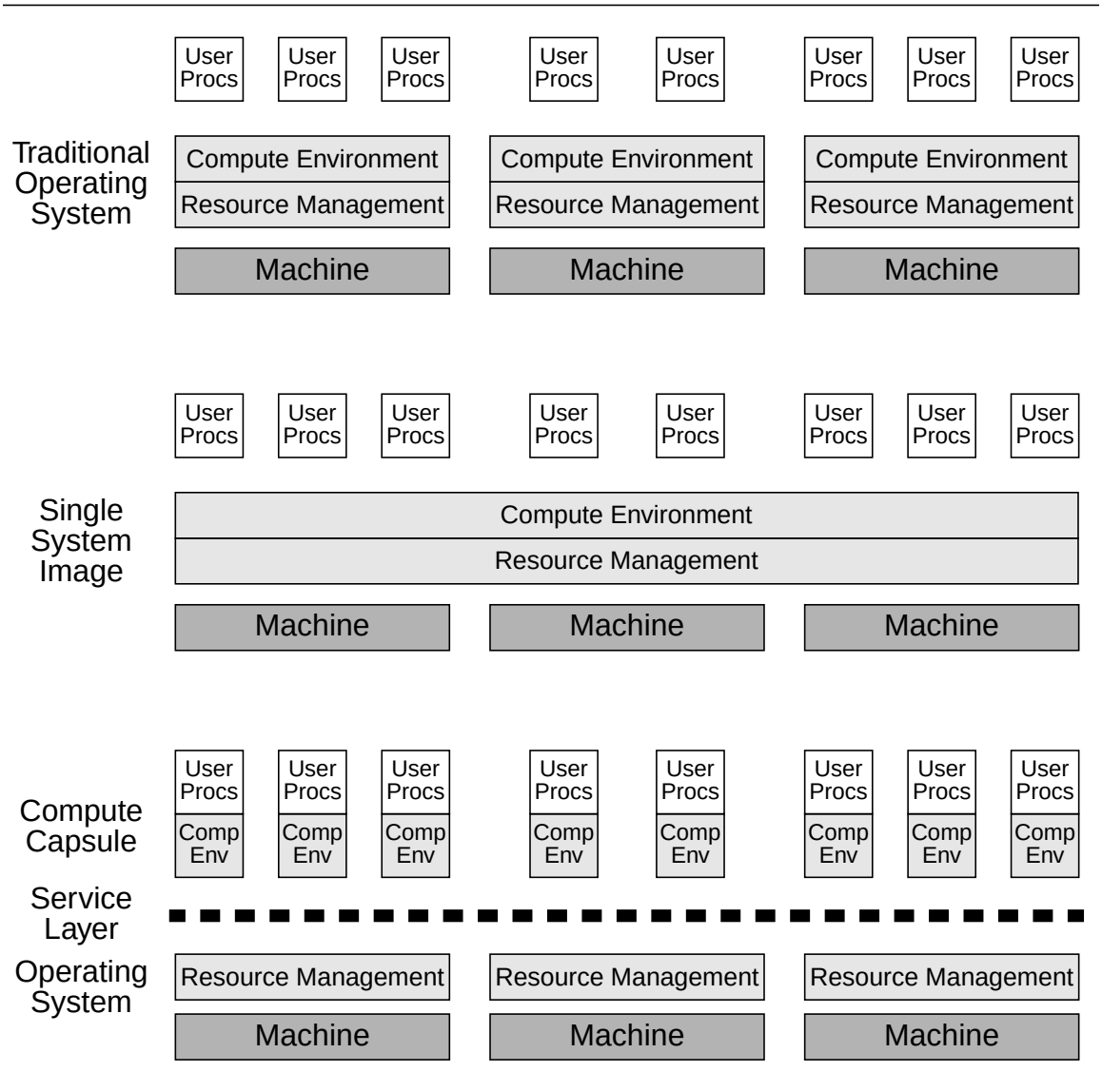


Figure 1-3 Comparison of traditional a operating system, a distributed operating system with a single system image, and a system with compute capsules.

provides no mechanisms for privacy, isolation, guaranteed performance, or customization. In our capsule-based approach, however, the machines function independently for the most part and require coordination only under exceptional conditions. Thus, a capsule system can easily be scaled to a large number of processors.

1.3.2.3 Capsule Benefits

Compute capsules have many desirable properties, and the following is a summary of the various benefits that they provide to the systems that utilize them.

- *Administration Costs*. System administration dominates the total cost of owning a computer. Consolidating resources and managing them centrally reduces this cost. Capsules leverage the management of the underlying system and thereby take advantage of these savings.

- *Host Independence*. Users merely want access to a computing/application service and should not need to concern themselves with unnecessary details, such as machine names and addresses, machine architecture, and the local environment. By providing a private interface to system resources, capsules ensure that applications are ignorant of any such host-specific details.

- *Mobility.* As users move, they would like to access their computing session without loss of continuity. Thus, sessions should migrate along with the user, or they should move onto a mobile computer for disconnected operation. Because capsules are self-contained and persistent, such migration is trivial.

- *Privacy and Isolation*. Shared computing environments offer no separation between users, but the private namespaces of capsules prevent users from observing or manipulating the resources in use by others. Additionally, because capsules can be assigned resources directly, the system can ensure performance isolation.

- *Customization*. Centralized computing offers users limited ability to customize their environments because of potential effects on the entire system. For example, users cannot install software packages that modify system directories. Because capsules provide a private, modifiable view of a shared file system, users can modify arbitrary files without the expense of providing each person with a separate and complete file system image.

- *Scalability.* The self-containment property of capsules enables a scalable system architecture that consists of mostly independently operating machines. In addition, the ability to move capsules to secondary storage supports a virtually unlimited number of persistent sessions. Without this capability, system resources could eventually all be consumed by an accrual of login sessions, even though many of them are idle.

- *Reliability and Availability.* In a centralized computing system, downtime of one machine affects many users. Because compute capsules are persistent computing environments, they can survive machine failures, and they can be re-located for operating system or hardware upgrades.

- *Dynamic Load Balancing*. Because compute capsules are self-contained, they can be easily moved between hosts in the system to provide a dynamic load balancing facility.

- *Ease of Deployment*. The proposed model can be easily deployed because existing applications need no special re-coding, re-compilation or re-linkage to run in a capsule and because compute capsules are implemented by simple extensions to commodity operating systems.

1.3.2.4 Applications of Compute Capsules

While this work was motivated by the creation of a scalable, ubiquitous computing environment, the fundamental concept of a compute capsule has many more applications. It can be used as a new approach to managing our computing environments today, as well as to improve the way we build future systems.

Customized User Environments

The various characteristics of compute capsules make the concept immediately useful in managing our computing environments today, as capsules can provide a variety of customized environments. For example, capsules with different levels of trustworthiness can be created. A capsule which is restricted to running a fixed application, with limited

access to the file system, and with no external connections could be used for a public kiosk or a guest user. A capsule could also be used to create an environment in which resource requests are monitored to watch for denial of service attacks or other suspicious activity (e.g. modifying certain system files).

Users often want environments with different personalities, each of which could be contained within a capsule. For example, a user could have one capsule for normal desktop usage, a fail-safe capsule with a minimal environment and no external dependencies, capsules for work-related and personal activities, etc.

Consider a shared project workspace, such as a development environment or a cooperative group communication system. The project coordinator can create a capsule and grant access rights to the group members. A private project file space can be created, and it will only be accessible to group members. Persons working on the project simply join the group capsule and begin to immediately share resources with the other members.

In our experience, many difficulties in software development and distribution stem from improper configuration of the computing environment and the unavailability of the correct versions of various tools and libraries. Such problems can be eliminated by encapsulating a complete working environment and the necessary tools in a capsule.

Debugging software is often problematic because many errors are non-deterministic. By periodically checkpointing the full state and logging intermediate operations, capsules can freeze a running program and capture all the events leading up to the bug. The capsule can be restarted and the events replayed in various orders to determine the cause of the error.

It is not uncommon that we find the need to reboot the operating system every now and then to either fix a problem or to upgrade the system. Instead of requiring all users to log off, a system using capsules can simply ask the users to tolerate a short pause while it saves the capsules to disk, reboots the system, and restarts the capsules.

System Architectures

Capsules can be used to provide centralized computing on a network of machines for a university or company campus, while giving the users the benefit of having a dedicated

computer. Users can gain access to their persistent login sessions on shared workstations, and they can customize their environments. In the meantime, the administration is centralized, thereby freeing the users from maintaining their own environments.

Capsules enable users to take home with them their computing state at work. They can access all their active processes and their graphical interfaces, without having to re-establish the working context, which may not be possible. Migrating the capsule between work and home allows fast interactive response without requiring a high bandwidth connection. The compute capsule may be transferred to the home machine during the commute. Similarly, this same idea can be applied to support traveling between company campuses across the world.

Capsules may be used by application service providers to offer a global, robust environment that is highly available, maintainable, and reliable. Capsules provide the framework for checkpointing active computations for fault tolerance, as well as for off-loading active computations to perform on-line maintenance or upgrades. Because these companies often host multiple accounts on the same physical machine, privacy and isolation between accounts is imperative, and users will demand a minimum guarantee in performance. Capsules can be used to meet these needs.

Finally, system administration for home machines has been a barrier to entry for many, and a great source of frustration especially for non-computer-professionals. By configuring the home machine as just another node that caches and executes capsules, the system administration effort can be centralized and provided by professionals.

1.3.2.5 Feasibility of Compute Capsules

This research introduces the concept of compute capsules and studies the issues and mechanisms surrounding the restructuring of operating systems to support them. Capsules are new systems architecture components that achieve the properties described above by splitting traditional functionality with the operating system. We demonstrate how to make this split so that capsules are host-independent, self-contained, capable of being customized, and subject to resource management and security policies, while leveraging

the underlying system as much as possible. To assess the feasibility of our design, we implemented a proof-of-concept prototype capsule system within the context of the Solaris 7 Operating Environment, and we demonstrated its use by encapsulating a wide-range of real-world applications running in a complete commercial systems setting. For example, our system can suspend, migrate, and resume full graphical user login sessions. To analyze the performance of our system, we quantified the overhead it introduces into normal operation, as well as the costs for managing and relocating capsules. In addition to the design, implementation, and evaluation of capsules, we also evaluated how capsules could be used to support a campus computing lab.

1.4 Dissertation Overview

This dissertation is organized as follows. In Chapter 2, we describe the design of our system, including the SLIM architecture for decoupling the human interface from the computational resources and the compute capsule abstraction. We continue in Chapter 3 with a discussion of the Sun Ray implementation details and an extensive evaluation of its interactive performance. Chapter 4 presents the details of our compute capsule implementation, as well as our experience with the prototype system. We compare our system with related research in Chapter 5. Finally, we summarize our work and present some conclusions and directions for future research in Chapter 6.

2 Remote Computational Service Architecture

The goal of this work is to move us closer to a portal-based computing model, in which users can gain global access from any network interface device to their active computations that reside in a pool of anonymous server resources embedded in the network. Toward this end, we have designed a new remote computational service architecture based on two guiding principles: (1) use modern powerful networks to separate the human interfaces from their computing resources, and (2) use self-contained active computations to support personal computing in a globally distributed and shared environment. As described in Chapter 1, a system based on these principles has many desirable properties and can serve as the foundation for our future computing infrastructure.

In this chapter, we present the rationale and design for our new system architecture. We begin by discussing how to decouple human interface devices from their computing resources using the SLIM system. This part of our architecture provides a means of replacing desktop computers with simple access devices for attaching to remote computations. Although we would eventually like to access our computing environments using any network-attached device, such as a mobile phone or personal digital assistant, the small screen sizes of the these devices require applications to employ fundamentally different user interfaces to achieve a reasonable look and feel, i.e. simply displaying standard applications onto such devices results in a poor user experience. In this work, we are concentrating on the infrastructure, not the programming model, and so we have limited ourselves to a standard desktop display device. Below, we discuss the components of a SLIM system and how they interoperate.

Next, we shift our focus from the desktop access devices to the necessary server-level support by describing the compute capsule abstraction. We discuss how capsules embody

personal computing environments within a shared context, as well as how they support self-contained execution. This provides the foundation and design rationale for compute capsules, and we then outline the system interface for manipulating them.

In addition to the design and use of compute capsules within individual hosts in the system, we require a thin distributed service layer (see Figure 1-3) to coordinate capsule activity across hosts. This layer is responsible for assigning and maintaining capsules names, coordinating capsule relocation, resource discovery, etc. This work, however, is concerned with the basic capsule functionality and provides only rudimentary behavior for the service layer. Additional support is beyond the scope of this research and is left for future work.

2.1 The SLIM Design

In this section, we describe the design and rationale for each of the components in the SLIM architecture: the interconnect fabric, the SLIM protocol, the consoles, and finally the servers (see Figure 1-2).

2.1.1 Interconnection Fabric

In a SLIM system, raw display updates are transmitted over the network to display devices. Thus, the SLIM interconnection fabric (IF) is the centerpiece of the architecture, and yet it is perhaps the simplest component. It is defined to be a private communication medium with dedicated, high-bandwidth connections between the desktop units and the servers. Such functionality can easily be provided by modern, off-the-shelf networking hardware.

For example, the preferred embodiment of the IF in a Sun Ray system is switched, full-duplex, 100Mbps Ethernet to the desktop, and a 1Gbps link to the server, which may support up to a few hundred users. The IF is defined in this manner so that the system can make response time guarantees and thereby provide high interactive performance, regardless of loading conditions. As we will later see in Chapter 3, however, these requirements can be significantly relaxed while still providing good interactive performance. Although it is possible to share bandwidth on the IF with non SLIM protocol traffic, care must be taken to ensure that response time does not suffer as a result. In

addition, there is no need to provide higher level (e.g., Layer 3 and above) services on the IF, nor the complex management typically provided on LANs.

2.1.2 The SLIM Protocol

The SLIM protocol takes advantage of the fact that the display tends to change in response to human input, which is quite slow. Thus, instead of refreshing the monitor across the IF, we achieve considerable bandwidth savings by refreshing the display from a local frame buffer and transmitting only pixel updates. Although the display is refreshed locally, it represents only soft state, which may be overwritten at any time. The full, persistent contents of the frame buffer are maintained at the server.

Command Type	Description
SET	Set literal pixel values of a rectangular region.
BITMAP	Expand a bitmap to fill a rectangular region with a foreground color where the bitmap contains 1’s and a background color where the bitmap contains 0’s.
FILL	Fill a rectangular region with one pixel value.
COPY	Copy a rectangular region of the frame buffer to another location.
CSCS	Color-space convert rectangular region from YUV to RGB with optional bilinear scaling.

Table 2-1 SLIM protocol display commands.

The protocol consists of a small number of messages for communicating status between desktop and server, passing keyboard and mouse state, transporting audio data, and updating the display. The display commands are outlined in Table 2-1. They compress pixel data by taking advantage of the redundancy commonly found in the pixel values generated by modern applications. For example, the BITMAP command is useful for encoding bicolor text windows, the FILL command is useful for painting solid regions, the COPY command is useful for scrolling or opaque window movement, and the CSCS command is primarily a multimedia optimization, which is described below.

In contrast, most other remote display protocols (e.g. X[39] and ICA[6]) send high-level commands (e.g. “display a character with a given font, using a specific graphics context”), which require considerable amounts of state and computation on the desktop unit. By encoding raw pixel values, the SLIM protocol represents the lowest common denominator of all rendering APIs. To take advantage of this protocol, a system need only change the device drivers for the human interface devices so that they use the SLIM protocol instead of direct device manipulation. In this way, applications can run completely unchanged with no porting cost, and SLIM consoles can interact with server machines of any architecture.

Another feature of the SLIM protocol is that it does not require reliable, sequenced packet delivery, whereas other remote display protocols (e.g. X[39], ICA[6], and VNC[47]) are typically built on top of a reliable transport mechanism. All SLIM protocol messages contain unique identifiers and can be replayed with no ill effects. Since the preferred IF implementation is a dedicated connection between console and server, errors and out-of-order packets are uncommon. In addition, the system can track the geometry of display updates and selectively send damage repair messages. Thus, systems using the SLIM protocol can be highly optimized with respect to error recovery. For example, repeated messages are ignored, and late messages are dropped if they draw to a portion of the display that was updated by more recent display commands, i.e. ones with higher sequence numbers. The SLIM error recovery scheme allows for more efficient recovery than packet replay and avoids “stop and wait” protocols.

2.1.3 SLIM Consoles

A SLIM console is a nameless terminal that consists mainly of a frame buffer attached to the interconnection fabric. It speaks the SLIM protocol on the IF to receive display primitives, decode them and hand off the pixels to the graphics controller. Similarly, it packetizes mouse and keyboard input into SLIM protocol messages and transmits them to the server. This allows us to make the desktop unit a cheap, interchangeable, fixed-function device. It is completely stateless and runs neither an operating system nor any applications, and the performance a user experiences is largely independent of the desktop hardware. This means that there is no need to upgrade the hardware of a SLIM console, unless the

display requirements undergo a fundamental change. In addition, the console is amenable to implementation as a simple ASIC, which could be embedded in a display device.

2.1.4 SLIM Servers

In the SLIM architecture, all processing is performed on a set of server machines which may include a variety of architectures running any operating system with any application software. These servers neither replace nor remove the standard set of servers common today, e.g. file servers, print servers. They merely consolidate and reduce the computing resources that were previously on desktops.

We only need to add to these servers a small set of system services: SLIM protocol drivers for human interface devices and daemons for authentication, session management, and remote device management. The authentication manager is responsible for verifying the identity of desktop users; the session manager redirects the I/O for a user's session to the appropriate console; and the remote device manager handles peripherals attached to the system via a SLIM console.

Using a SLIM system is straightforward. A user may connect a SLIM console to the interconnection fabric and power it on. At this point, it has no network identity and does not know the identity of any SLIM session servers. So, it broadcasts a query on the IF, and a server will respond and establish a connection with the console. Now, the SLIM console is ready for operation. The user presents identification to the terminal, which transmits it to the authentication manager at the server. Once the user is authenticated, the session manager will search for the user's active computing session and instruct the SLIM device drivers to direct their I/O on the IF to the appropriate terminal, and the user can interact with the session on the console in the same way as if it were the display, mouse, and keyboard attached directly to the server. When done working, the user can disconnect from the console, and the session manager responds by instructing the SLIM device drivers to cease transmission over the IF. The sessions remains active and running until the user attaches again later.

2.1.5 The SLIM Programming Model

Although the SLIM protocol is intended to be used primarily by low-level device drivers, applications may wish to utilize the protocol directly in a domain-specific manner. For example, we have implemented a video playback utility and a 3-D game by converting each display frame to YUV format and using the CSCS command to transmit the data directly to the console. Although these applications could function as regular X clients, this approach is more efficient and produces higher quality results. To support this programming model, we developed an API for the SLIM protocol. The API includes operations for querying the session manager to find the identity of the terminal to which protocol messages should be sent, as well as session geometry to know which portions of the display may be drawn. Applications must generate appropriate display data in a format compatible with the SLIM protocol, but the API includes methods for packetizing pixel data into SLIM messages and transmitting them across the IF with appropriate sequencing. Error recovery is handled by the application, and the API provides means for the programmer to learn which messages have been lost and to respond in an application-specific manner, e.g. for video the correct action may be to ignore lost messages since the display will be completely overwritten by the next frame. To control traffic shaping on the IF, the API includes methods for querying bandwidth and response time statistics, as well as for predicting future usage requirements (based on historical consumption). This enables applications and SLIM consoles to coordinate network traffic so that bandwidth is distributed fairly.

2.2 Design of Compute Capsules

Compute capsules are private, portable, and persistent computing environments. Their principle constituents are active processes, and membership is fully dynamic, i.e. a capsule can contain an arbitrary collection of processes and their associated system environment. This includes privileges, shell histories, environment variables, working directories, assigned resources, installed software, etc. Members of a capsule are free to communicate and share with each other in any of the traditional ways, e.g. pipes, shared memory, signals, etc. Inter-capsule communication, on the other hand, is restricted to internet sockets and globally shared files. This helps simplify the encapsulation so that capsules can move

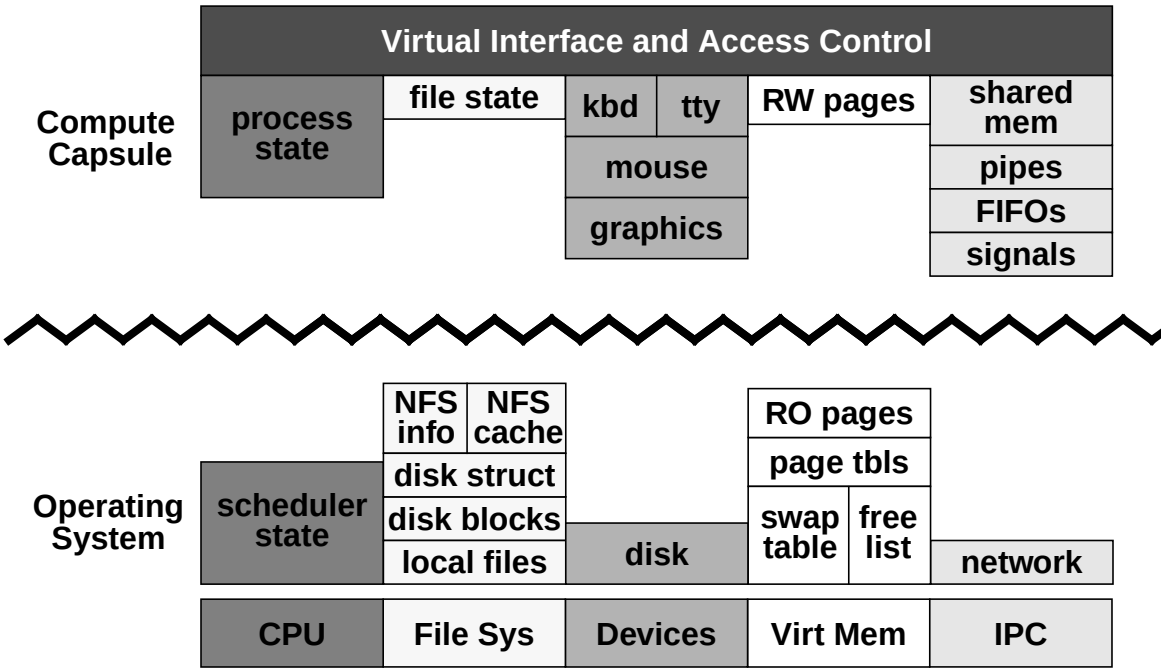


Figure 2-1 Re-partitioning of state ownership between the operating system and compute capsules.

without restriction. For example, a pipe opened between processes in different capsules would force both capsules to reside on the same machine. Although capsules may be arbitrarily bound to physical machines, a single capsule cannot span multiple machines. If this were not the case, capsule members would be restricted in the forms of IPC they could employ. The use of compute capsules is completely transparent, and applications need not take any special measures, such as source code modification, re-compilation, or linking with special libraries. Further, a system using compute capsules can seamlessly inter-operate with systems that do not.

Our philosophy for designing capsules is to re-partition operating system functionality and state so that host-dependent and personalized elements of the computing environment are moved into capsules, while leveraging policies and management of the shared underlying system, as illustrated in Figure 2-1. In this section, we present the key principles in the design of compute capsules, as well as some of the major technical challenges.

2.2.1 Embodiment of Personal Computing Environments

Compute capsules embody a user's personal computing environment. As such, we would like them to be isolated from each other, capable of being customized, and managed by the underlying system.

2.2.1.1 Private Namespace

Just as modern virtual memory subsystems provide separation between individual processes, capsules should offer privacy, isolation, and security within a shared environment. Current systems allow users to access all processes and other system resources, i.e. anyone can gather (and sometimes manipulate) information about the activities of others.

Compute capsules address this issue by providing a private namespace for its member processes. All name references are local to a capsule (i.e. they are valid only within the capsule), and capsule contents are not externally visible or accessible (unless they are explicitly exported). For each capsule, the underlying system is responsible for maintaining the allowable set of references to resource objects. This ensures that capsules cannot synthesize names of objects in other capsules, i.e. such references will return an error. Thus, capsules provide privacy and isolation because they are completely disjoint such that it is impossible for an object in one capsule to name an object in another capsule. Further, by controlling the set of resource objects in the namespace, the system can create a secure environment within a capsule. For example, the system could create a capsule that lacked the ability to name various sensitive system files.

2.2.1.2 User Customization in a Shared Environment

Compute capsules operate within a shared computing infrastructure, and they should leverage the underlying system as much as possible. At the same time, users should be able to customize the environment a capsule provides. However, shared systems are managed by privileged administrators, and users do not have the necessary access rights to modify the system configuration. For example, users cannot install software, override remote file

mounts, prevent name clashes with the files of other users, or control access to their personal computing environment.

Our approach is to split administration functionality between the operating system and compute capsules. Although the system administrator is responsible for managing access to system resources, the owner of a capsule has complete control over access rights to the capsule itself, i.e. which processes and users may join. This enables owners to manage the security of their capsules independently of privileged system users.

In addition, each capsule has a personal view of the underlying (global) file system. Capsules map portions of the file system into their namespace, thereby leveraging the underlying system for most file administration tasks (e.g. standard software packages, backups, etc.). The file system view is private to a capsule and fully modifiable by its owner, and it provides a copy-on-write mode of file access to support the modification of read-only or system files. In addition, capsules may export portions of their view to other capsules to support data sharing.

2.2.1.3 System Management Units

A compute capsule is a first-class object within the operating system, which means it is an embodiment of resource management and security policies for its set of member processes. This enables the system to treat groups of processes as single units for accounting and resource management decisions, which has three important effects. First, when resources are needed and available, they can be easily accessed by assigning them to a particular capsule. This may require migrating a capsule between machines, thereby dynamically load-balancing the system. Second, users may demand a guaranteed level of resources or performance isolation from others. This can be achieved by assigning a minimum, fixed portion of resources directly to capsules. Third, capsules may be subjected to policy constraints. For example, the system may impose restrictions or quotas on their resource allocations, and capsules could be assigned different levels of trust, e.g. restricting network access or certain file system operations.

Traditional operating systems, on the other hand, manage resources at the level of individual processes. This approach is too fine-grained because it treats all requests uniformly and does not provide a means for isolating the performance effects of multiple users. Whereas current systems control security and administration policies at the level of individual users, capsules provide a means for applying policy decisions to arbitrary collections of processes. This gives greater flexibility and improved control over system management.

2.2.2 Self-Contained Units of Execution

Compute capsules fully encapsulate active computing environments. They are free to move between any pair of binary compatible machines, or they may be disconnected from any physical machine and suspended in persistent storage. To achieve this functionality, capsules must access the underlying system in a host-independent fashion, and they must include all the state necessary to be completely self-contained.

2.2.2.1 Host-Independent Naming

Unfortunately, current operating systems do not provide a clean separation between the kernel and user-level code at the resource interface. Tokens used to identify resources frequently encapsulate the binding to the underlying physical resource as well, and resource tokens are often treated as globally unique when in fact they are only unique for a specific machine. This creates unnecessary host dependencies that irreversibly couple active processes to particular instances of an operating system on particular machines. For example, a Unix process identifier (PID) is a token by which one process may refer to another process. It is actually an index into the active process table inside the kernel, i.e. it is the physical resource itself, which is far from a clean abstraction. A PID is only valid on a particular machine and is not globally unique. Thus, a process cannot move to another machine and maintain the same PID.

Compute capsules address this problem by providing a host-independent interface to underlying system resources. Each capsule assigns virtual tokens to all resources. The tokens are unique to each capsule and therefore valid across all machines in the system. They are transparently and dynamically bound to different physical resources on any host

in the system. In this way, capsules separate the naming of resources from their physical embodiment, and a mapping from the virtual capsule namespace to the physical resource namespace is made for each object reference a capsule makes (and vice versa).

To help solidify these concepts, consider again the example of process identifiers (PIDs). Each process in a compute capsule is assigned a virtual PID, which is mapped to the system-level PID on the underlying machine. Processes can name other processes only by their virtual PIDs. If the capsule moves to a new host, its member processes may be assigned new system-level PIDs, but the virtual PIDs will remain the only valid process names within the capsule.

2.2.2.2 Host-Independent State

To be fully self-contained, capsules must incorporate the complete state of the active computing environment for its member processes. This information is normally spread throughout the system, but capsules compartmentalize the state associated with a group of running processes into a single unit for better manageability. In this way, capsules represent a new organizational unit to encapsulate a user's personal computing environment within the system architecture, just as the process abstraction is an organizational unit for the components of a running application.

As mentioned earlier, much of the critical state of active processes is maintained within the kernel and is inaccessible to the application. Such state is dispersed among various kernel data structures, and there is no facility for extracting it, storing it off-line, or re-creating it from an intermediate representation. For example, the complete state of a Unix pipe cannot be precisely determined by user-level routines. The identities of the endpoints and any in-flight data are known only by the kernel, and there is no mechanism to duplicate a pipe only from its description.

Our approach is to add a new interface to the operating system so that capsules can import and export critical kernel state, i.e. the operating system merely caches the state. Although the kernel manages all state information, ownership has been re-partitioned between capsules and the operating system. Compute capsules own user-specific and

host-dependent information, while the underlying system maintains global state (see Figure 2-1). Each kernel module that maintains process state, such as the Solaris fifo module that maintains pipe and fifo state, is enhanced with two complementary functions: (1) export state for a particular resource object in a portable representation, and (2) import a representation of resource state and recreate it exactly. In the case of an open pipe, this state would include the processes on the endpoints, the control settings, streams modules serving as filters, and any in-flight data. However, the state object is an opaque data type outside the module that implements pipes.

2.3 Operation of Compute Capsules

Capsules allow users to benefit from the shared administration of centralized computing while enjoying the freedom to customize their environments. At the same time, capsules are designed so existing applications can run transparently without modification, which is critical for capsules to be adopted and used in practice. In this section, we describe how capsules meet these goals by discussing their high-level operation. We start with the creation and maintenance of capsules and then focus on two important aspects of the system: the private file system view and relocating capsules. To support these operations, we defined new operating system interface routines, which we list in Table 2-2.

`capsule_create`	Create a new capsule and install a process.
`capsule_join`	Move a process into an existing capsule.
`capsule_acl`	Manage the access control list for a capsule.
`capsule_checkpoint`	Suspend a capsule and record its state.
`capsule_restart`	Restart a capsule from its recorded state.

Table 2-2 New capsule-related operating system interface routines.

2.3.1 Capsule Management

Managing capsules includes operations to create and join capsules, as well as mechanisms for administering capsule environments, both internally and by the underlying system.

2.3.1.1 Capsule Directory Service

Although this work does not focus on the distributed service layer to coordinate activity across capsule systems, the capsule directory service is one component of this layer that is necessary for normal capsule operation. The capsule directory service is a well-known network information repository, i.e. an LDAP-like network database. The capsule directory is responsible for maintaining compute capsule status information. In particular, it maintains a logical name for a capsule, its virtual network address (see Section 2.3.4), and its current location (IP address of active host or file system location if suspended). All capsules can query the directory service to discover this information for other capsules.

2.3.1.2 Creating and Joining a Capsule

An application may instantiate a new capsule or join an existing capsule via the two new system calls, `capsule_create` and `capsule_join`, respectively. Typically, a user would perform these operations via standard login facilities, which have been modified to capture entire user sessions as capsules.

The `capsule_create` request includes the name of the user who will own the capsule and the initial process to be executed (e.g. a login shell). Normally, the capsule will have the same owner as the invoking process, which requires no special permission. Creating capsules for other users, however, requires privileged access. Once the system verifies the access rights of the requestor, it allocates a unique identifier and instantiates the capsule on the machine of its choice, perhaps based on load-balancing demands.

Initialization includes the following operations:

(1) The creation of a view of the file system that is unique to the capsule and based on user preferences and system defaults (Section 2.3.2),

(2) The initialization of all name translation tables required to virtualize the system interface (Section 2.3.3), and

(3) The inclusion of the initial process.

The initial process may be the activation of an arbitrary application, but system-provided creation facilities may offer only a standard interface, such as launching a user shell or window session from the login program. Capsule membership is inherited by the descendants of a process, i.e. children of a member process will belong to the same capsule, regardless of user identity.

Once the capsule has been successfully created, the system registers its name and location with the well-known capsule directory service. This enables users and applications to locate, manage and join other capsules. When all members of a compute capsule exit, the system reaps the capsule itself. The resource name translation tables are freed; the file system view is removed; and the capsule name is removed from the directory service.

The only way to enter another capsule is for a process to execute a `capsule_join` operation. The requestor queries the capsule directory service for the name of the target capsule and requests that the system create a new process in the specified capsule. The requestor's access rights are checked (Section 2.3.1.3), and then the desired application is instantiated within the target capsule. Typically, the remote login facility is modified so that it can either instantiate a new capsule or issue `capsule_join` to enter an existing capsule.

2.3.1.3 Owner Control of Internal Capsule Environment

Maintaining the internal environment of a capsule is the responsibility of its member processes. This includes managing the file system view, process control, customization, etc. Because any member process may perform such operations (when permitted by the underlying system), access to the capsule must be carefully controlled. Thus, each capsule has a private list of users who are granted access to manage the state of the capsule. This list can only be modified by the owner of the capsule via the `capsule_acl` system call. A user who is not on the list will not be permitted to create a process in the capsule, and there are no specially privileged users. This ensures that the members of a capsule cannot exceed their privileges by executing a program that assigns additional rights. For example, the Unix administrative user, root, does not have permission to join arbitrary capsules.

Thus, although a Unix setuid program may assign administrator-level access rights, it will not enable a process to illicitly access other capsules.

2.3.1.4 Access to the Underlying Operating System

The underlying operating system is responsible for caching compute capsules and managing capsule services. To administer system policies or to manipulate capsules themselves, we require access to the operating system from outside the scope of capsules. We provide this ability via a special administrator login session that is identical to the computing environment on non-capsule systems, i.e. it provides a “back door” into the underlying host. Processes in this session can view and manipulate all processes and resources in the system, regardless of capsule membership.

2.3.2 File System View

Whereas a compute capsule resides entirely within a single machine at any given time, the underlying file system is globally available. This provides a simple mechanism for sharing data between capsules, as well as for maintaining a coherent view of the operating environment as the capsule moves. For example, a user may own multiple capsules that access a shared database, or software packages can be made globally available for use by all capsules.

Each capsule maintains a private view of the global file system. This view is the primary mechanism for user customization, and it is similar to the per-process name space of Plan 9 [43]. When a capsule is created, a new file system view is synthesized by creating a root-level directory and mapping elements from the global file system into it. Once the file system view has been created, the system alters the environment of the compute capsule so that it will access files relative to the root of this view. This enhances both security and privacy, as files outside the view cannot be referenced after this point, i.e. capsules are restricted to accessing only files within their private views. Protection of files mapped into different views in different capsules is provided by the underlying file system.

By default, file system views are automatically populated with system directories and files necessary for normal user operation: application binaries, libraries, certain device drivers,

user home directory and data files, etc. The contents of the view may be arbitrarily configured by the owner of the capsule, which means that system-level files that have been mapped into the view could be modified, e.g. the Windows system registry. Because these files are shared by other capsules, they are mapped into a file system view in a copy-on-write mode. When a capsule modifies such a system file, a private copy is added to the capsule, and the complete altered file is added to the capsule so that the changes persist, even if the capsule is relocated. In this way, changes only affect the capsule that made the modifications. Files and directories normally assumed to reside in local storage (e.g. the Unix swap directory where temporary files are often created) must also move with the capsule to maintain the appropriate access semantics. Conversely, there are many standard, machine-local directories that are the same across systems (e.g. /bin contains application programs on Unix systems), which may be mapped into the file system view directly from the local machine for improved performance.

The file system view provides capsule owners with the ability to customize their capsule environments within a shared system. Modifying the file system view merely affects its structure and contents, while the underlying system is leveraged heavily for normal file system administration (e.g. backups, maintaining software packages, installing disks, configuring file servers, etc.). While capsule owners may install their own software or modify system files, they will typically rely on professional system administrators for these tasks and simply map in the underlying files.

2.3.3 Capsule Relocation

An important feature of capsules is that they can be suspended and moved to any machine. This requires a host-independent interface to the system and self-contained state, and it must be transparent to capsule processes.

As mentioned above, capsules provide a host-independent and isolated computing environment via a private and completely virtual namespace. Each resource a compute capsule can name is assigned a virtual token that is only valid within the capsule. Resource objects include processes, shared memory segments, open devices, etc. A name translation table binds the user-visible tokens to the physical resources on the underlying machine to

which the capsule is bound. A name translator is interposed between user processes operating in the capsule virtual namespace and the underlying system, which operates on the physical namespace of the machine. Translation is transparent to the operating system and applications, i.e. they require no modifications.

Capsule relocation is provided by the `capsule_checkpoint` and `capsule_restart` system calls. When a user invokes `capsule_checkpoint`, the system suspends the capsule and records its complete state, which can then be used by to `capsule_restart` to resume the capsule elsewhere. These operations make use of the kernel state import/export interfaces to extract the necessary state of a compute capsule. In addition, the name translation tables are persistent within a capsule and are mapped to new machine-local values if the capsule is moved to another host, thereby providing transparent mobility of the computing environment. The file system view and any file contents maintained in the capsule are also included in the recorded state information.

2.3.4 Capsule Communication

As mentioned previously, the member processes of a compute capsule are free to communicate through any standard means, and the state of any open inter-process communication (IPC) channels will be recorded by `capsule_checkpoint` and re-created by `capsule_restart`. In this way, sets of processes communicating in arbitrary ways can be transparently moved between machines or suspended in secondary storage without loss of functionality. Because the processes in a capsule are suspended as a group, recording this state is an atomic and fairly straightforward operation, given the state import/export interfaces. However, inter-capsule communication is via internet sockets and requires additional effort because the remote endpoint will most likely not be simultaneously suspended, and we would like it to operate without modifying the existing network infrastructure. In this section, we present our approach for relocating open network connections between capsules, but the actual implementation is beyond the scope of this work.

When a capsule with an open external connection is suspended via the `capsule_checkpoint` operation, the system ceases outbound transfer and sends a

notice to the remote site instructing its network controller to pause physical packet transmission. To support connectionless protocols, the capsule must record the addresses and ports of all packets from external transmitters and then send messages to suspend their transmission until further notice. These pause messages trivially extend the network protocols to include a special packet to instruct a remote system to suspend transmission on a particular socket. Note that the network infrastructure remains unchanged; only the semantics of certain packets are treated specially by a capsule system. For example, UDP does not include a back channel, but this protocol layers one on top. Also, note that only the protocol stack is being changed on the host, not the application program.

The protocol stacks on the two endpoints then exchange a set of messages to agree on the state each has, i.e. which packets have been transmitted, received, and buffered. Once the two endpoints have synchronized to ensure that all transmission has been paused and that the sockets are in a stable configuration, the capsule can continue the checkpoint operation. The state of the open socket is recorded, along with any received data that has not been read by an application. In addition, the capsule registers its current status (suspended) and location (on disk) with the capsule directory service. Meanwhile, processes in the remote capsule may continue to transmit on the open socket. Eventually, the local write buffers will fill, and the processes will block until the buffer is drained. Similarly, reads on the socket will eventually block when there is no more data in the read buffer. Thus, the fact that the other end of the connection has been suspended on disk is completely transparent to the remote capsule.

Eventually, the capsule will be revived to the running state on a possibly new machine, and the remote capsule may have relocated as well. When the capsule is restarted, it registers its new status and location with the capsule directory service and queries it for the status and location of the remote capsule. If both capsules are running, their respective hosts will contact each other and re-establish the connection and drain any buffers. Processes can resume as if neither capsule has moved. Although this design is fairly straightforward, there are several issues that must be resolved for this operation to function correctly.

2.3.4.1 Capsule Addresses

To unambiguously transmit packets to a particular capsule (regardless of its location), it must have a canonical and immutable IP address. If capsules are not routable entities, then packet delivery would be based on the address of some host in the system. This can lead to confusion if the capsule is not executing on that host. It is not straightforward how to distinguish between packets intended for the capsule and packets intended for the host, and capsules could be assigned different addresses at different times. This complicates routing and can result in unpredictable behavior when an application tries to address a capsule.

Our solution is to employ a technique much like mobile IP [41] in which a mobile host maintains a "home address" and a "care-of address." The home address is a well-known and static IP address, and the care-of address is the current IP address of the mobile host on the network. Packets destined for the mobile host are always transmitted to the home address, where a "home agent" will forward them to the current care-of address. Similarly, capsules are assigned a fixed IP address, and all packets destined for the capsule must utilize it. We will discuss routing in the next section.

Because capsules may be created quite frequently, we would like a flexible means for assigning IP addresses to them, i.e. without requiring network administrator support. Thus, we can use a dynamic configuration protocol, such as DHCP [16], to maintain a pool of addresses for capsules and assign them as part of the `capsule_create` operation. This mechanism is fairly well understood, and standard implementations exist today that can be used directly.

2.3.4.2 Address Translation

Since a capsule is assigned its own IP address, applications can establish connections to capsules in a manner that is compatible with existing network protocols and infrastructure. However, the capsule will be bound to a physical machine with its own IP address, and we would like packets to be correctly routed without requiring additional support in the network. This issue can be resolved by extending the virtual namespace concept to IP addresses and port numbers, much like network address translators (NATs) [17] in modern routers and firewalls.

Whenever a host system receives a request to transmit data to a particular IP address and port that are associated with a process in a remote capsule, the system will map it to the port and IP address of the current host to which the process and its capsule are bound. A reverse translation is performed for all incoming packets. If there is no currently valid mapping, the capsule directory service is queried for the necessary translation. This simple tunneling concept ensures that open network connections can be transparently retargeted to a new host.

When a capsule is about to suspend itself, it notifies any hosts to which it has open connections (or which have transmitted data on a connectionless socket) that they should invalidate their IP address mappings for the capsule (and pause transmission). When the capsule is later revived, it registers its new host IP address with the capsule directory service and also sends it to the capsules it notified prior to suspension. Any such capsules that are suspended will re-establish their connection via a query to the directory service when they are revived.

2.3.4.3 Legacy System Support

If the remote host is a legacy system that does not support compute capsules, it will not support the relocation of the capsule to a new host with a new IP address. The system could spawn a forwarding agent on the current host of the capsule and assign it the incoming connection from the remote system. When the suspended capsule is later revived, it could notify the forwarding agent of its new location, and the agent would begin passing packets through to the new location. This is the approach adopted in the Sprite distributed operating system [40] to support migration of processes with open IPC channels. The downside to this approach is that the original host must remain active, and a capsule that migrates frequently may have many active forwarding agents.

Thus, we propose the use of specialized routers at the edges of networks that contain capsule systems. These routers perform the NAT function between legacy systems and capsules systems. When a capsule is suspended, the router tables are updated to indicate that the capsule cannot be reached. The router will adjust the flow control properties of incoming connections so that the sender stops transmitting, and the router will maintain the

connection by responding to keep-alive messages from the remote host. Packets using a connectionless protocol can be safely dropped. When the capsule is later revived, the router is updated with its new location, and packets are tunneled to the new host

Note that the same tunneling technique could be used for routing packets between capsules, i.e. the router performs the NAT function instead of the host. Which method of implementation to use is a matter of efficiency, complexity, and personal taste.

2.4 Summary

Our design of a remote computational service architecture is based on the notions of decoupling the human interface devices from their computing resources and encapsulating active computing environments in portable units. We have described the SLIM architecture for attaching the human interface devices (mouse, keyboard, and display) directly to a dedicated interconnection fabric that provides access to remote computing resources. SLIM consoles are stateless, cheap, fixed-function, interchangeable access devices that require zero administration and maintenance. They provide a means for accessing standard desktop applications from any network portal. We outlined the SLIM protocol for transmitting low-level device I/O across the interconnection fabric, and we discussed the server support necessary to provide service to the consoles.

The computational resources in our new system architecture are based on a common, shared infrastructure. This enables the system to amortize administration and hardware costs, as the servers will be professionally managed and shared across a large user population. In addition, users can attach to the global server pool from any location, thereby gaining access to virtually unlimited resources from the desktop, while simultaneously supporting user mobility. At the same time, however, the servers must support personal computing environments that are private and isolated from each other, host-independent and portable, persistent, and capable of being customized by the user.

Unfortunately, these goals represent a clash between the properties of dedicated and shared computing. In this chapter, we have presented the compute capsule abstraction, which addresses this problem by fully encapsulating active computing environments as a

system-level object. Capsules contain an arbitrary collection of active processes (and their associated state), and they serve as organizational units for the host-dependent and personalized elements of a computing environment. This ensures that capsules can enjoy the benefits of the personal computing model within the context of a shared environment. To demonstrate how capsules achieve these properties, we presented the design of the compute capsule abstraction, and we discussed the operation of capsules in practice, as well as the compute capsule user interface.

3 SLIM Implementation and Evaluation

The SLIM architecture extends the notion of thin-client systems to an extreme design that is amenable to implementation as a simple, stateless, fixed-function appliance. Sun Microsystems has developed an implementation of a SLIM system in a commercial product line, known as Sun Ray, and we present the details of that implementation in this chapter.

To assess the ability of the SLIM approach to meet the demands of traditional desktop computing, we conducted an extensive evaluation of the performance of the Sun Ray implementation, and we report the results of our analysis also in this chapter. Because SLIM consoles are simply human interface devices that perform no computing tasks whatsoever, the standard performance metrics are meaningless. Instead, we must focus on the user experience with such a system, i.e. the interactive quality of service. Thus, we present a quantitative analysis of the two questions critical to the success of the SLIM architecture: (1) the extent to which today's interconnect can support graphical displays of interactive programs, and (2) the resource sharing advantage of such a system. To answer these questions, we develop a measurement methodology for characterizing the interactive performance of a computer system and then use it to analyze Sun Ray. We consider the interactive performance for individual users during stand-alone operation, as well as multiple users on a shared session server. Finally, to explore the limits of the system, we analyze its ability to support high-demand multimedia applications, such as 3D video games.

3.1 The Sun Ray Implementation of the SLIM Architecture

The Sun Ray desktop appliance and supporting infrastructure are composed of inexpensive, commodity components available today. In this section, we present the details of the Sun

Ray 1 implementation [38][52], upon which our experiments are based. This includes the interconnection fabric, the SLIM protocol, the Sun Ray desktop unit, and the session servers.

3.1.1 Interconnection Fabric

The Sun Ray interconnection fabric is implemented on top of standard Ethernet and uses the UDP/IP network protocol for communication between the desktop unit and the server, i.e. the SLIM protocol is layered on top of UDP. Such functionality can easily be provided by modern, off-the-shelf networking hardware. Systems have been built that utilize cable modems, DSL lines, shared 10Mbps hubs, shared 100Mbps hubs, and 10/100Mbps switches from various vendors. The Sun Ray 1 desktop unit supports a 10/100 Base-T Ethernet connection, and we used switched, full-duplex 100Mbps Ethernet with Foundry FastIron Workgroup switches in our experiments. These switches support 24 10/100Mbps connections and a 1Gbps uplink to the server, and they have 4.2Gbps of internal switching capacity.

3.1.2 SLIM Consoles

The Sun Ray 1 enterprise appliance includes four major hardware components: CPU, network interface, frame buffer, and peripheral I/O. It utilizes a low-cost, 100MHz microSPARC-IIep processor with 8MB of memory, of which only 2MB are used. The network interface is a standard 10/100Mbps Ethernet controller, and the frame buffer controller is implemented with the ATI Rage 128 chip (with support for resolutions up to 1280x1024 at a 76Hz refresh rate and 24-bit pixels). A four-port USB hub is included for attaching peripherals, including keyboard and mouse.

In addition, the desktop unit includes a smart card reader for authentication. Upon insertion of the smart card, a user’s session will immediately appear on the console display. When the card is removed, the session is detached from the terminal but remains active on the server. The console also includes an audio codec and I/O connectors for microphone and speakers, as well as an input connector and video digitizer for capturing composite video.

The firmware on the console simply coordinates the activity between the components, i.e. it moves data between the interconnection fabric and appropriate interface devices. Because the console can merge display streams from multiple sources, the firmware includes a dynamic bandwidth allocator, which ensures that high-bandwidth streams do not starve other streams for console service. This traffic shaper requires display streams to embed bandwidth requests, which it uses to schedule available capacity. In addition, the firmware gives feedback to the session concerning the actual bandwidth granted and consumed. Bandwidth is allocated by giving preference to small requests and then dividing left-over bandwidth fairly among the large requests.

3.1.3 SLIM Servers

To implement the SLIM protocol on the Sun Ray session servers, the virtual device driver for the X-server was modified to transmit interface I/O across the interconnection fabric. This device driver controls access to the frame buffer, mouse, and keyboard, and it is implemented beneath all user-level software, which enables all X applications to run completely unchanged.

In addition to the X-server virtual device driver, an audio driver has been implemented to exchange acoustic information with the desktop units. This driver implements the standard audio interface for all Sun audio devices and thus seamlessly supports all audio-based applications.

Sun Ray consoles are zero administration devices, but they must operate on conventional networks, which require them to have IP addresses for routing and packet delivery. Rather than involve system administrators to assign static IP addresses to the consoles, the Sun Ray server implements a DHCP service so that consoles can obtain dynamic addresses without any user intervention.

The authentication manager and session manager have also been implemented. These service daemons work together to coordinate activity between the consoles and the server. When a user inserts a smart card or logs in to a desktop console, the console sends a message to the authentication manager, which verifies the identity of the user and queries

the session manager for any active sessions belonging to the user. If there is no active session, a new login session is started; otherwise, the session manager directs the SLIM protocol generators in the active session (e.g. the virtual device driver in the X-server) to connect to the user's current console. When the user removes the smart card or logs out, the session manager instructs the SLIM protocol generators to pause transmission to the console.

Finally, a library that implements the SLIM programming model described in Section 2.1.5 has been implemented. This library is primarily used by application developers of high-performance multimedia programs, and it provides direct access to the SLIM protocol in a manner that maintains an appropriate relationship with the authentication and session managers.

3.2 Methodology for Analyzing Interactive System Performance

Characterizing processor performance is fairly well understood, and a wide range of industry standard, throughput-oriented benchmarks exist, e.g. the SPEC benchmark suite. However, because the SLIM console is a simple access device, we are concerned with interactive quality of service, but little is known about analyzing the interactive performance of complex systems. In this section, we discuss the issues that make such analysis difficult, present our evaluation methodology, and discuss related techniques.

This work was done in cooperation with the research group at Sun Microsystems Laboratories that developed the recently announced Sun Ray 1 enterprise appliance product, which is an embodiment of the SLIM architecture. The prototype and the product were used by the developers (over 150 engineers, managers, marketing personnel, and support staff) as their only desktop computing device for the past two years, and they have found their interactive experience to be indistinguishable from that of working on the high-end, workstation-class machines to which they were accustomed. This research attempts to quantify the interactive performance of the SLIM architecture scientifically, using the Sun Ray 1 prototype and product as a basis for the experiments. We focus on the workgroup environment where SLIM consoles are connected to the servers via a dedicated,

commodity network carrying only SLIM protocol traffic. In particular, our experiments were performed with dedicated, 100Mbps switched Ethernet.

Analyzing interactive system performance is fundamentally different than traditional throughput-based approaches. Response time is the critical metric, and so stress tests that measure how quickly the system can perform certain tasks are of limited use because any response that arrives in less than 100–150ms appears instantaneous to humans. For example, a typical graphics benchmark may test how quickly the system can render a window containing a particular bit pattern. However, such a test gives no indication of what the user's response time will be like. How long does it take from the time a user clicks a button until the display update appears on the screen? Would the user notice? To answer these kinds of questions, we must characterize the activity at the human interface.

3.2.1 Characterizing the Human Interface is Difficult

The SLIM architecture is based partly on the observation that the display is typically updated in response to user activity. We expect humans to have a relatively slow rate of interaction, and so the display should be quiescent much of the time, thus requiring modest resources to provide good performance. To verify this observation, we must analyze the interaction and display patterns of modern applications during active use to determine if SLIM can meet their demands.

To analyze interactive performance by characterizing the human interface first requires us to select a set of benchmark applications to study. There are no such industry standard benchmarks, and care must be used to ensure that the results will be meaningful. Applications should be chosen not based on their computational, memory, or I/O demands. Instead, they should exhibit a wide range of input and display characteristics, as well as responsiveness demands. Because we are studying the human interface, only graphical applications are meaningful. Once benchmark applications have been selected, several issues still remain: developing tools to collect information, how to measure the applications, and how to provide realistic input to the benchmarks.

Current systems do not provide the appropriate measurement tools to quantify response time. Available tools (such as netstat, vmstat, or iostat) aggregate critical metrics across all processes and users active in the system. For example, the network bandwidth consumed by an individual process cannot be determined, as the system merely reports the total traffic through the network interface card in the machine. In addition, there are no tools for measuring the activity at the human interface to determine what portions of the display are changing and how frequently, as well as the characteristics of user input.

To collect accurate and precise data on response time at the human interface requires the instrumentation of each application to be tested. Measuring the elapsed time from the moment a user triggers an input event to an application (e.g. clicking the mouse on an element of the graphical user interface) to the instant at which the application has completed servicing the event and produced a result (e.g. displaying a dialog box) is highly application-specific. Although this approach produces the best results, it is impractical. Source code is often not available, and adding correct instrumentation to a large, complex program, such as the Netscape web browser, is an error-prone and time-consuming task that requires a great deal of semantic knowledge about the program.

To obtain useful results, the benchmark applications must be measured while being driven by a set of realistic input, which matches actual human behavior as much as possible. One means for addressing the issue of providing program input is the use of scripts of application-level events. Such scripts interface with the application using a well-known interface and generate timed events from a stored list. For example, Windows provides a scripting macro language to automate tasks in programs, such as MS Word or Outlook. Unfortunately, such scripts usually do not reflect actual user behavior. They are often used in a batch mode in which events are replayed as quickly as possible with none of the “think time” delays a human would exhibit, or they use delays between events that are chosen without an underlying model of human behavior, which frequently results in rates that are too coarse-grained with respect to human interaction [11][12][59]. Thus, this method of input generation produces results that cannot easily be translated into overall system performance.

A better means of simulating actual user behavior with scripts is to record a series of input events from an actual user session and play them back to a specific application, which reacts as if a user were presenting it with live input. For example, Windows and X-based systems include utilities for recording input events at the protocol level and then replaying them as part of an animated presentation. While this technique more closely resembles real usage patterns, the timings are based on the initial recording and do not adapt to the situation that exists during playback. For example, the recorded input stream may indicate that a mouse click should be generated at a particular time. The intent is for the mouse event to trigger an action on an interface widget. If the widget has not yet been displayed, the resulting behavior is undefined. Thus, we cannot experiment with different loading conditions or other scenarios with this approach.

3.2.2 Our Approach

Using the Sun Ray implementation of the SLIM architecture, we analyzed the interactive performance of the system, and our evaluation methodology consists of four steps:

(1) Establish the basic performance of each component (the interconnection fabric, session servers, and consoles) of the SLIM system using stand-alone tests.

(2) Characterize GUI-based applications by their communication requirements, evaluate how well the SLIM architecture and protocol support such applications, and compare the results with the X protocol.

(3) Evaluate the opportunity for sharing offered by the SLIM architecture by measuring its interactive performance under shared load and also by obtaining load profiles of actual installations of the system.

(4) Measure the limit of SLIM by evaluating its performance on high-demand multimedia applications.

To ensure that our results would have the widest applicability, we wanted to select a set of application benchmarks that cover a wide range of resource demands, interactive characteristics, and display patterns. Further, they should be applications with graphical

interfaces, since that is the modern class of applications with which users interact directly. Thus, we have chosen a set of commonly-used, GUI-based applications to use as a benchmark suite, and we list our selected applications in Table 3-1.

Category	**Application**
Image Processing	Adobe Photoshop 3.0
Web Browsing	Netscape Communicator 4.02
Word Processing	Adobe Frame Maker 5.5
PIM	Personal Information Management tools (e.g. e-mail, calendar, report forms)
Streaming Video	MPEG-II decoder and live video player
3-D Games	Quake from id Software

Table 3-1 Benchmark applications.

We instrumented the SLIM protocol driver used by these applications so as to measure the I/O properties at the human interface. Recall that this driver is implemented as a virtual device driver beneath the X-server. To obtain results that reflect real-world user behavior, measurements were taken by having a group of 50 people perform their normal work-related tasks with the benchmark applications. The users separately ran each application in a dedicated fashion for at least ten minutes, and a very lightly loaded server was used to ensure that measured performance would reflect individual users in isolation. During these user studies, all SLIM protocol commands and system resource usage was logged for each application session. Because we employed actual user trials to provide realistic input to the test applications, we avoid the problem of trying to mimic human behavior through artificial means. In addition, the SLIM protocol logs enable us to precisely determine when user input events occurred and when display updates appeared on the screen, thereby eliminating the need to instrument applications to gather such information.

As mentioned above, gaining precise statistics on per-process resource consumption is not possible with current systems. Therefore, we created a measurement tool that interfaces

directly with the kernel to collect detailed information on the activities of each process in the system, including bytes transmitted and received on the network, bytes read or written to disk or network files, processor utilization, physical memory occupancy, etc. All collected information is time-stamped with a high-resolution timer. This tool adds only negligible overhead to a running system, thereby allowing us to collect detailed statistics for each active job in the system.

3.2.2.1 Single-User Systems

The profiles collected during the user studies fully characterize the human interface and system requirements for each application during normal use. Thus, using the profiles, we can obtain the human input rate (which corresponds to the display update rate for most applications), the cost for the input-induced display updates (i.e. response time), and the overall bandwidth and resource requirements. This gives us the ability to quantify how well the system performs from the perspective of a single user on a stand-alone system.

3.2.2.2 Multi-User Systems

To characterize interactive performance under shared load, we use an indirect method. We create a highly interactive application with a fixed and regular resource requirement, and instrument it to measure the response time its user experiences. We run this application with different system loads and use its measured latency as a yardstick to gauge the responsiveness of the overall system. When the yardstick application performs well, we can infer that the system would perform well for other interactive applications.

To simulate multiple active users we use load generators to “play back” the resource profiles that were recorded during the user studies. These load generators merely mimic the resource consumption of the recorded profiles, i.e. they do not run an application with a script of input events. Thus, they easily adapt to different load levels on the test system. This approach has the advantage of providing a consistent metric across systems and applications, and it addresses the drawbacks of script-based approaches, which cannot adjust to loading conditions.

3.2.3 Related Techniques

As mentioned above, there is little work which is directly relevant to the analysis of interactive system performance. In this section, we consider techniques that are currently used for quantifying response time in a single-user, stand-alone system, as well as a system with multiple active users.

3.2.3.1 Single-User Systems

Endo *et al.* have conducted some preliminary work at Harvard University in which they attempt to measure the interactive performance of a general system [18]. The primary focus of their work is to argue for the use of interactive event latency (i.e. response time) metrics to evaluate system performance, which is similar to our own measurement goals. They instrumented Windows NT versions 3.51 and 4.0 as well as Windows 95 and recorded processing latency for each input event for a set of stand-alone applications. Their goal was to quantify the amount of dedicated CPU time the system spends servicing each input event, and they present service time distributions for some simple applications.

The main difficulty they experienced is that it is impossible to know exactly how much processing time is needed to service an event without instrumenting each application. So, they instrumented the idle loop and relied on properties of the event queue to deduce the status of the running job, i.e. when it was idle waiting for I/O and when it was idle waiting for user input. Because the event queue is shared by all running jobs, they could not study more than one simultaneously active application and distinguish between them. In contrast, our indirect benchmark technique enables us to evaluate multiple active applications.

In addition, their approach is aimed at quantifying the interactive performance of stand-alone desktop systems, such as a PC. However, we are interested in evaluating remote display systems, which requires us to take into account both the network overhead and the remote terminal service time. Their characterization of response time only considers the event service time on the processor, which corresponds to the work done by the session server in a SLIM system. Thus, their approach can be used to complement our measurement system, which focuses mainly on the performance effects that the SLIM decoupled architecture introduces.

3.2.3.2 Multi-User Systems

Capacity planning is a commonly used evaluation technique that determines the number of simultaneously active users a server can support [11][56][59]. Such studies are useful for making resource allocation decisions, but they do not adequately assess interactive performance because they tend to take a long-term, coarse-grained view of system activity, thereby losing information on interactive performance. In addition, user activity is modeled in one of three ways, all of which have their drawbacks. First, canned scripts or macros are frequently used, but they have no interactive delays and thus merely measure throughput. Second, recorded scripts with interactive delays are used to emulate users, but timing dependencies between the client (e.g. application) and server (e.g. the X-server) make such emulations only valid when the system is in underload. Finally, queueing theory is used to model users based on average resource profiles. While this approach has the advantage of being able to leverage well-known mathematics techniques to make precise predictions, it cannot model the system in overload.

3.3 Evaluation of the SLIM Architecture

Using the experimental methodology outlined above, we evaluated the interactive performance of the SLIM architecture, and we present our results in this section. We begin our analysis with a stand-alone characterization of each major component in the system. Then, we proceed with characterizations of interactive performance of single-user systems during dedicated operation, as well as multi-user systems during shared operation. We conclude with an exploration of the multimedia capabilities of a SLIM system.

All our experiments are based the Sun Ray implementation of the SLIM architecture. This study began before the product was released, and as a result some experiments were conducted using prototype Sun Ray desktop units. The only difference between the prototype and the production version is the use of a slightly upgraded graphics controller. Thus, we repeated any experiments conducted on the prototype unit for which this modification could have had an impact. In practice, the only significant changes occurred in the raw protocol processing performance of the desktop, which is a throughput measure that is primarily useful for ensuring that future versions of the product do not regress in

performance. Depending on the nature of the experiments and machine availability, we employed a variety of session servers during our study, but all experiments were performed using full-duplex, 100Mbps Ethernet and Foundry FastIron Workgroup switches. We summarize our test configuration for each experiment in Table 3-2.

Experiment	Desktop Unit	Server				
		Model	Processor(s)	RAM	Swap	OS
IF response time	Sun Ray 1 prototype	Ultra 2 workstation	1 296MHz UltraSPARC-II	512MB	1GB	Solaris 2.6
x11perf	Sun Ray 1	Enterprise E4500	8 336MHz UltraSPARC-II	6GB	13GB	Solaris 2.7
User studies	Sun Ray 1 prototype	Ultra 2 workstation	2 296MHz UltraSPARC-II	512MB	1GB	Solaris 2.6
Processor sharing	Simulated	Enterprise E4500	10 296MHz UltraSPARC-II	4GB	4.5GB	Solaris 2.7
IF sharing	Same as server	Ultra 2 workstation	2 296MHz UltraSPARC-II	512MB	1GB	Solaris 2.7
Multimedia tests	Sun Ray 1	Enterprise E4500	8 336MHz UltraSPARC-II	6GB	13GB	Solaris 2.7

Table 3-2 Hardware configurations for experiments on interactive performance. In all cases the interconnection fabric was 100Mbps Ethernet with Foundry FastIron Workgroup switches. All machine hardware is manufactured by Sun Microsystems, Inc.

3.3.1 Performance of SLIM Components

Although we are primarily concerned with the user experience on a SLIM system, it is also useful to obtain baseline measures of the raw performance of the individual components. This helps characterize the fundamental performance limits of the system. We conduct a stand-alone analysis of the interconnection fabric, server graphics performance, and protocol support on a Sun Ray 1 desktop unit, and the results are summarized in Table 3-3 and Table 3-4.

Benchmark	Result
Response time over a 100Mbps switched IF	550 µs
x11perf / Xmark93	3.834
x11perf / Xmark93 — no display data sent on IF	7.505

Table 3-3 Stand-alone benchmarks for the Sun Ray 1.

3.3.1.1 Response Time Over the Interconnection Fabric

Response time is a major concern for any remote display architecture. Other approaches, such as X Terminals and Windows-based Terminals, process input events locally on the desktop. A SLIM system, however, must transmit all input events to a server and then wait for it to send back the results. Humans begin to notice delays when latency enters the 50–150ms range [53]. To provide a high-quality interactive experience, it is crucial that response time remain within these bounds.

Typical local area networks are shared and carry a variety of traffic. Depending on the current load, latency can be highly variable, and it is difficult to make response time guarantees in such an environment. We ensure that latency remains within the prescribed limits by using a switched, private network for the interconnection fabric. In the SLIM architecture each desktop unit has a direct connection to the servers, and only SLIM protocol traffic is carried over it. Since there is no interference or other outside effects, the added latency is exactly the round-trip delay over the interconnect.

To demonstrate the effectiveness of the SLIM approach, we wrote a simple server application that accepts keystrokes from a SLIM console and responds by sending characters to the console. We measured the total elapsed time from the instant a keystroke is generated at the SLIM console to the point at which rendering is complete and the pixels are guaranteed to be on the display.

With the test setup shown in Table 3-2, the elapsed time was found to be 550µs. In contrast, if we type the characters in an Emacs editor, as opposed to our simple application, the delay is found to be 3.83ms. Clearly, in such an environment the communication medium is a negligible source of latency, and the end result is that the response time that users

experience is effectively dependent on the processing time on the server. Users are, therefore, unable to distinguish between the response times observed from a remote SLIM console and a monitor connected directly to the server itself.

It is interesting to note that on a stand-alone workstation, the service time is much higher at 30ms [37]. This is most likely due to buffering of input in the keyboard device driver so that block data transfers can be used. Such an optimization is not required for network I/O since it can be memory-mapped directly into a user's address space. Thus, the SLIM implementation can actually be more responsive than a dedicated workstation.

3.3.1.2 Server Graphics Performance

A key factor in the display performance of any thin-client system is how well the server can generate the protocol. In the Sun Ray 1 implementation of the SLIM architecture, the display driver for the X-server is responsible for translating between pixel data generated by X primitives and the SLIM protocol, as well as transmitting display update commands over the IF. To analyze the graphics performance of a SLIM server, we ran the SPEC GPC x11perf benchmark.

The x11perf benchmark is outdated and no longer used, but it provides a useful indication of performance. We ran the benchmark on the system shown in Table 3-2, and we used the Xmark93 script to generate a single figure of merit from the results. The X-server achieved a rating of 3.834. Although the X-server must interpret the commands and transmit display data to the console, its Xmark value is quite comparable to values for X terminals, which receive commands and interpret them locally. For example, the NEC HMX reported a value of 4.20 to SPEC. If we eliminate the actual transmission of SLIM protocol commands, the Xmark performance rating of the X-server improves to 7.505, indicating that network operations represent a significant performance factor for this benchmark.

3.3.1.3 Protocol Processing on the Desktop

The performance-limiting factor on a SLIM console is the highest sustained rate at which it can process protocol commands. To determine this limit, we created a server application that transmits sequences of protocol commands to a Sun Ray 1 console up to the point

where the terminal cannot process the transmitted commands and begins to drop them. Once these sustained rates were determined for various command types and sizes, we calculated the protocol processing cost in terms of a constant overhead per command as well as an incremental cost per pixel. The results are presented in Table 3-4.

Protocol Command	Start-up Cost	Cost per Pixel
SET	5000 ns	270 ns
BITMAP	11080 ns	22 ns
FILL	5000 ns	2 ns
COPY	5000 ns	10 ns
CSCS (16 bits/pixel)	24000 ns	205 ns
CSCS (12 bits/pixel)	24000 ns	193 ns
CSCS (8 bits/pixel)	24000 ns	178 ns
CSCS (5 bits/pixel)	24000 ns	150 ns

Table 3-4 Sun Ray 1 protocol processing costs.

The SET command has a fairly high incremental cost because pixels must be expanded from packed 3-byte format to 4-byte quantities suitable for the frame buffer. On the other hand, the FILL and COPY commands are very inexpensive. The BITMAP command has a high start-up cost in order to set the state of the graphics card and must be amortized over a large number of pixels to be of the most benefit. The color space convert and scale (CSCS) command not only has a high start-up cost in order to configure the graphics controller, but it also has a fairly high cost per pixel. Still, it is more efficient than the SET command when fairly large numbers of pixels are sent, which is usually the case for applications that utilize image or video data. Also, it provides a significant reduction in bandwidth, making it worthwhile despite the high processing overhead.

3.3.2 Interactive Performance of Single-User Systems

Equipped with an understanding of the raw performance of a SLIM system, we now turn to consider the interactive experience in a single-user configuration. To characterize the I/O

requirements of interactive programs, we use the GUI-based benchmark applications listed in Table 3-1, i.e. Photoshop, Netscape, Frame Maker, and PIM. As mentioned in Section 3.2.2, we employ user trials to gather data during real-world use of the applications. The data were collected on two identical systems, as listed in Table 3-2. Each server had a 1Gbps uplink from its IF and ten terminals attached to it. The servers were completely underloaded at all times, and users had the impression that they were working on a dedicated workstation. Thus, the gathered traces are indicative of stand-alone operation.

To obtain these results, we instrumented the SLIM protocol driver in our implementation of the X-server. All X and SLIM protocol events were recorded and timestamped. The logging overhead is not measurably significant and therefore does not perturb the results. These logs enable us to determine input and display characteristics as well as network traffic requirements.

Our analysis proceeds as follows. To determine system response time for our test applications, we begin by analyzing the rates of human interaction. Because the display tends to change in response to user activity, the input rate corresponds to the display update rates. Next, we characterize the sizes of the updates (as well as the compression benefit afforded by the SLIM protocol), the cost to transmit them over the network, and the protocol processing costs on both the console and server that are necessary to complete the display update. This gives us a measure of the response time for each display update, which we compare against the thresholds of user perception. Next, we evaluate the overall bandwidth requirements of the SLIM protocol and compare them with the requirements of the X protocol. We also analyze how well the SLIM protocol scales down to lower bandwidths, and we conclude with a characterization of the memory requirements on the server.

3.3.2.1 Human Input Rates

Based on the data gathered in the user studies, we first analyzed typical rates of human interaction. We defined input events to be keystrokes and mouse clicks. All input events are transmitted to the server for processing, and mouse motion events do cause some display updates for these applications (e.g. rubber banding). However, such motion events

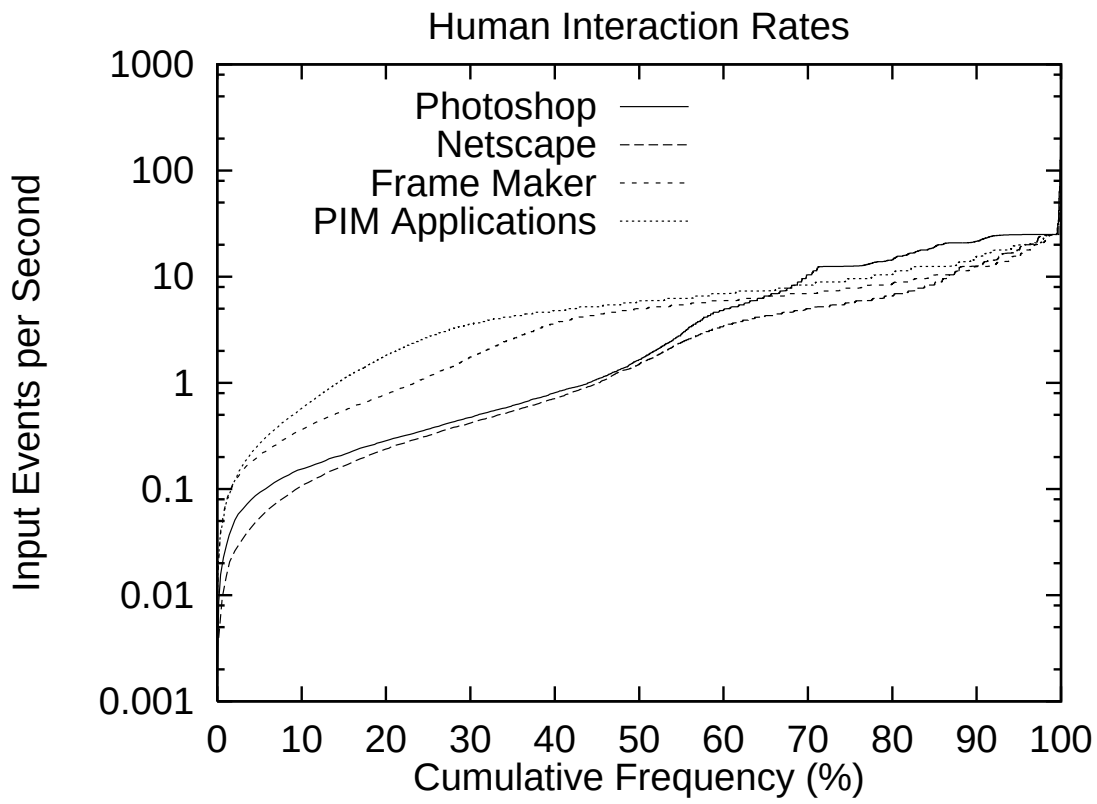


Figure 3-1 Cumulative distributions of user input event frequency. Input events are defined as keystrokes and mouse clicks. Histogram bucket size is 0.005 events/sec.

are demarcated by a button press and release, which serve to toggle the drawing mode. In these cases the motion-induced display updates are more naturally regarded as a single update caused by the initial button press. For each of the GUI-based benchmark application session profiles, we calculated the frequency of input events, and Figure 3-1 presents the results.

The most important thing to note in this graph is that human input rates are very slow, which implies that the display will be changing very slowly as well. In particular, we see that for each application, less than 1% of input events occur with frequency greater than 28Hz. This is important because it indicates an application-independent “upper bound” on the rate at which humans generate input. Also, roughly 70% of all events occur at low frequencies, i.e. less than 10Hz (or more than 100ms between events). Despite the diversity of these applications, the interaction patterns are quite similar. Even so, Netscape and

Photoshop tend to be much less interactive than Frame Maker or PIM, as indicated by their substantially larger percentage of events occurring at least one second apart.

3.3.2.2 Pixel Update Rates

Next, we consider the pixel changes induced by human input. Correlating input events to display updates can only be done by instrumenting all applications. As mentioned in Section 3.2 however, source code is frequently unavailable, and adding correct instrumentation to a complex program, such as Netscape, is prone to errors. Therefore, we use the following heuristic to estimate the correlation between input events and display updates: all pixel changes that occur between two input events are considered to be induced by the first event. Although this is not true in all cases, it provides a close enough approximation for the applications in this experiment over the course of the ten-minute user sessions. To obtain these values, we sum the number of pixels affected by SLIM display primitives recorded between the events, and we present the results in Figure 3-2.

This graph depicts the cumulative distributions of pixels altered per input event. All displays used in these experiments had a resolution of 1280x1024 pixels (i.e. 1.25Mpixels). The important thing to note is that display updates typically affect only a small fraction of the full display area. For example, nearly 50% of all input events for any application cause less than 10Kpixels to be modified. Further, only 20% of Frame Maker or PIM events affect more than 10Kpixels, and only 30% of Netscape or Photoshop events affect more than 50Kpixels. By way of comparison, a shell window with 80 columns and 24 rows of characters may require roughly 160Kpixels. Also, note that Netscape is more demanding than Photoshop, but as we will later see, its compressed bandwidth requirement is much lower.

This result, coupled with the rates of human interaction, indicate that (for this class of applications) the contents of the display change only slowly and moderately over time. This is important because it has a significant effect on the SLIM architecture, namely that even as display requirements increase over time, human input rates (and therefore required update rates) are unlikely to change. Thus, it is unlikely that SLIM consoles would need to be upgraded for users of these types of applications.

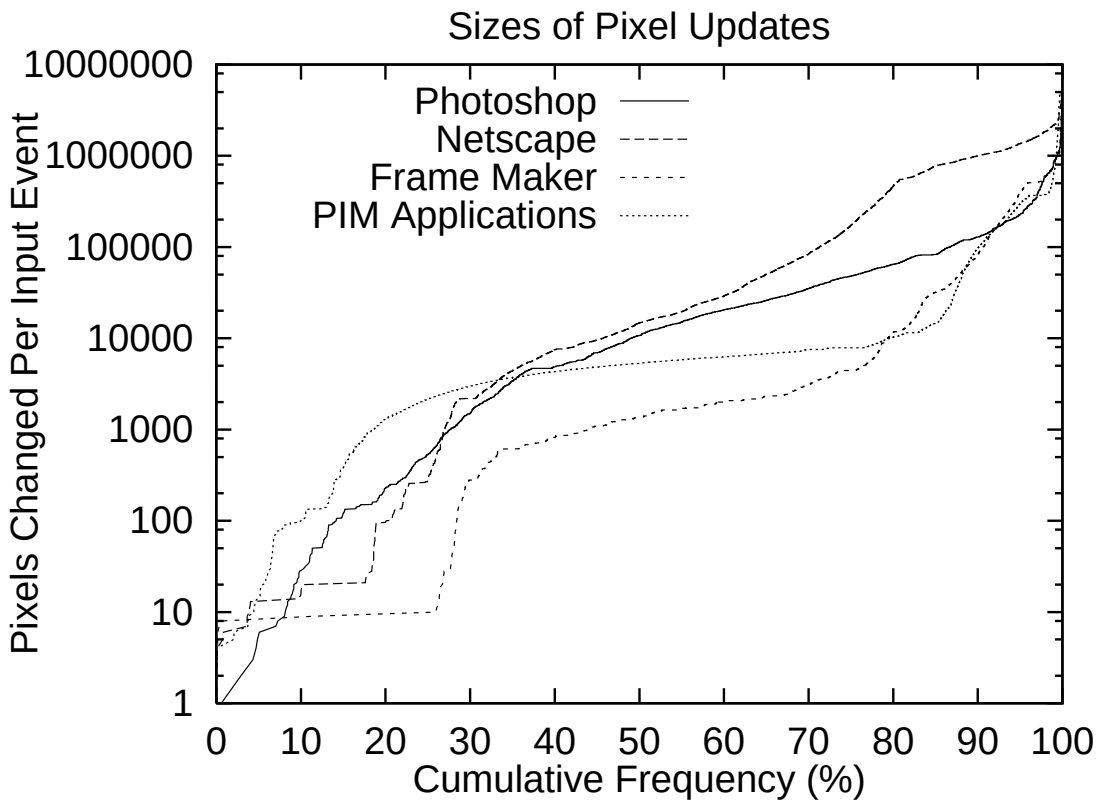


Figure 3-2 Cumulative distributions of pixels changed per user input event. Input events are defined as keystrokes and mouse clicks. Histogram bucket size is 1 pixel.

3.3.2.3 Compressed Pixel Update Rates

To assess the effectiveness of the SLIM pixel encoding techniques, we analyze the compression factor afforded by each of the SLIM display commands. In addition, we examine the sizes and network transmission delays of display updates once they have been compressed with the SLIM protocol.

In Figure 3-3 we present the data reduction afforded by each of the SLIM protocol display commands for the GUI-based benchmark applications. The important observation to make is that the protocol provides a factor of 2 compression for Photoshop and a factor of 10 or more for all other applications. The FILL command is extremely effective, reducing bandwidth by 40%–75% across the applications. The BITMAP and COPY commands are utilized to different extents, but they also provide substantial compression. PIM and Frame

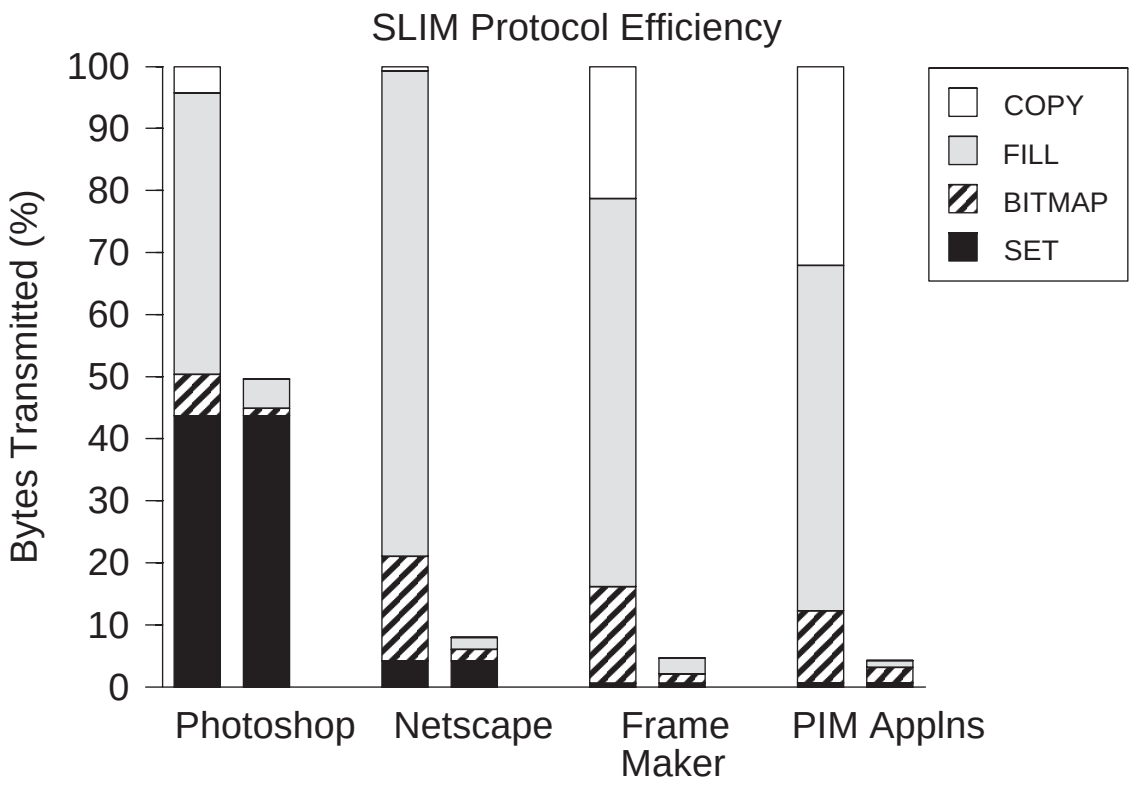


Figure 3-3 Efficiency of SLIM protocol display commands. The left-hand bar in each set depicts uncompressed data, and the right-hand bar depicts use of the SLIM protocol. CSCS is not used in these benchmark applications.

Maker enjoy particularly large savings because they employ a great deal of bicolor text and scrolling. The SET command represents pixel data which cannot be compressed via the SLIM protocol, but only Photoshop has a substantial portion of such commands. Thus, we can see that the protocol has done a good job of compressing the pixel data where possible.

In Figure 3-4 we present the cumulative distributions of the amount of SLIM protocol data transmitted per user input event for the GUI-based benchmark applications. From this graph we can see that (over a 100Mbps interconnection fabric) the transmission delays will be quite small. For example, even a large update of 50KB incurs only 3.8ms of transmission delay. More specifically, only 25% of Photoshop and Netscape events require more than 10KB to encode the display update, and only 5% require more than 50KB. Frame Maker and PIM have even lower requirements; only 17% of events require more than 1KB for

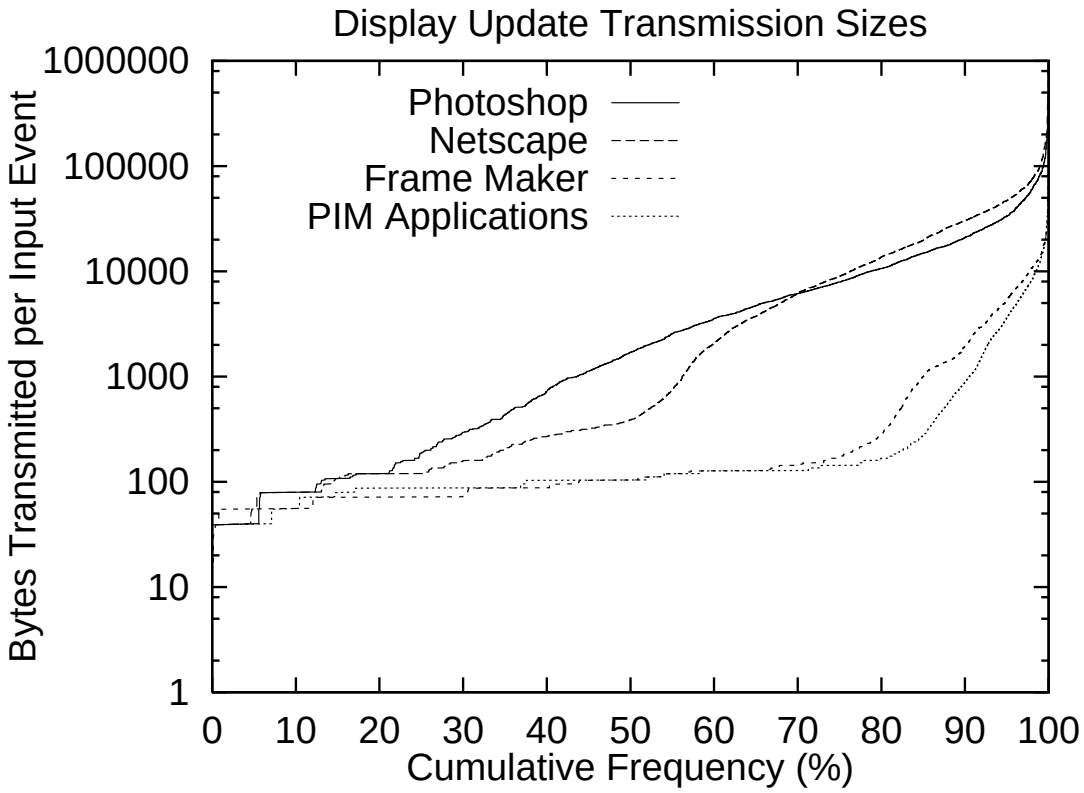


Figure 3-4 SLIM protocol data transmitted per user input event.
A cumulative distribution is depicted for each application. Input events are defined as keystrokes and mouse clicks, and the histogram bucket size is 1 byte.

display updates, and only 2% require more than 10KB. Thus, display updates will incur just a few milliseconds of transmission delay.

3.3.2.4 SLIM Protocol Command Statistics

To gain an understanding of how the different test applications utilize the SLIM protocol, we compiled statistics for each display command to determine what fractions they contribute toward the total number of commands executed, number of pixels rendered, and number of bytes transmitted. Our results are presented in Figure 3-5.

As we can see from the chart, the FILL command is the most heavily utilized, accounting for the majority of commands and pixels rendered. However, its contribution toward actual bytes transmitted is a much smaller fraction, indicating its effectiveness at compressing the display stream. The COPY command is so seldom used that it is not visible in the chart of

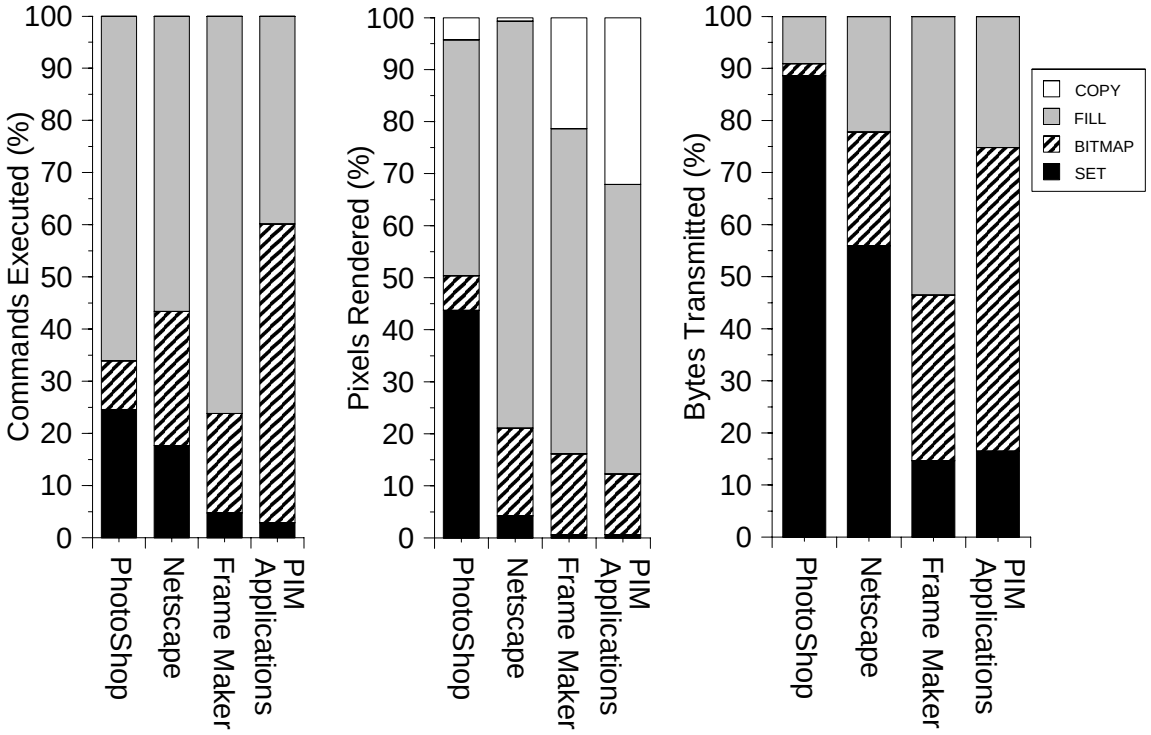


Figure 3-5 SLIM protocol command statistics.

commands executed, accounting for less than 0.1% of commands. However, it causes a significant number of pixels to be updated while consuming very little of the bandwidth — a highly effective optimization. The BITMAP command is used to varying degrees but accounts for a relatively small fraction of pixel updates. The SET command represents pixel data that cannot be compressed with the SLIM protocol. The applications use it to a fairly minor degree; only Photoshop and Netscape (which display images) use it substantially, and only Photoshop renders a significant amount of pixels with the command. However, in all cases the SET command accounts for substantial bandwidth, particularly Photoshop. To enhance the protocol, more sophisticated techniques would need to be applied to the pixel updates that fall back to use the SET command, but the potential savings in bandwidth would be offset by an increased computation overhead.

3.3.2.5 SLIM Protocol Processing Costs

A key goal of SLIM is to minimize the resources on the desktop. However, reading protocol commands from the network and decoding them for display incurs processing overhead.

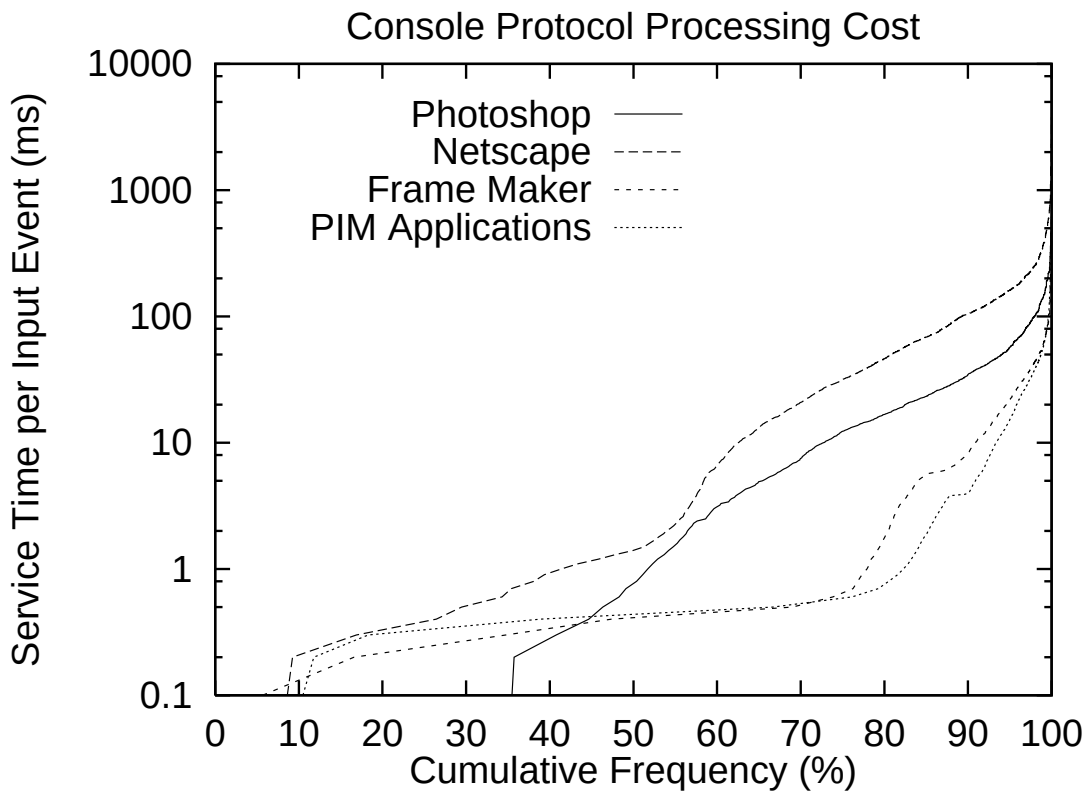


Figure 3-6 Cumulative distributions of display update service times. Measurements were made on the Sun Ray 1 console, and the histogram bucket size is 0.1ms.

The CPU is mostly idle until a display command arrives, followed by a burst of activity which entirely consumes the processor. To analyze this cost, we monitored the service time on the Sun Ray 1 console for the SLIM protocol commands sent as part of each display update during the user studies described above. In Figure 3-6, we report the cumulative distributions of these service times for display updates.

The important thing to note in this graph is that response time is almost always below the threshold of perception, i.e. in 80% of all cases service time is below 50ms. A small fraction of service times exceed 100ms and may be noticeable, but note from Figure 3-3 that there are correspondingly large display updates, for which human tolerance is typically higher.

The total service time for a display update is the combination of these console service times and the network transmission delays discussed in Section 3.3.2.3. As we can see, the network adds little extra cost, and so the total service time is quite short. Thus, we conclude

that the simple, low-performance microSPARC-IIep is more than adequate to meet the demands of these applications. In fact, an even lighter weight implementation could possibly be used, and a combined processor and memory implementation of the SLIM console could provide exceptional performance at an extremely low cost point.

Finally, although we have focussed on the cost of decoding protocol messages on the desktop, the server also incurs overhead due to encoding pixel values and generating protocol messages. However, this overhead is extremely low, accounting for a mere 1.7% of the X-server's total execution time when servicing the benchmark applications. Thus, we conclude that the added cost of protocol processing to send the pixel updates is marginal compared to the savings in bandwidth it affords.

3.3.2.6 Average Bandwidth Requirements

To put the results from this section into perspective with respect to other thin-client architectures, we calculated the network bandwidth requirements for our set of benchmark applications under three protocols. In Figure 3-7, we present the average network bandwidth required for each application using the X protocol, the SLIM protocol, and a simple protocol in which all the changed pixels are transmitted (labelled "Raw Pixels" in the chart).

There are two important points to observe in this graph. First, it is quite interesting to note that the X and SLIM protocols have similar bandwidth requirements. X performs slightly better on the Frame Maker and PIM applications, which were, in fact, the classes of programs for which X was optimized. However, this is insignificant because the bandwidth requirements of these programs are so low. On the other hand, Photoshop and Netscape represent a new class of applications in which image display is the common operation, and they require an order of magnitude more bandwidth. In these cases, SLIM outperforms X, and the absolute size of the bandwidth differential is substantially larger than for the other applications (note log scale in the graph). We thus conclude that the SLIM protocol is at least competitive with X in terms of bandwidth requirements, while providing a much simpler, lower-level interface that requires substantially fewer computing resources.

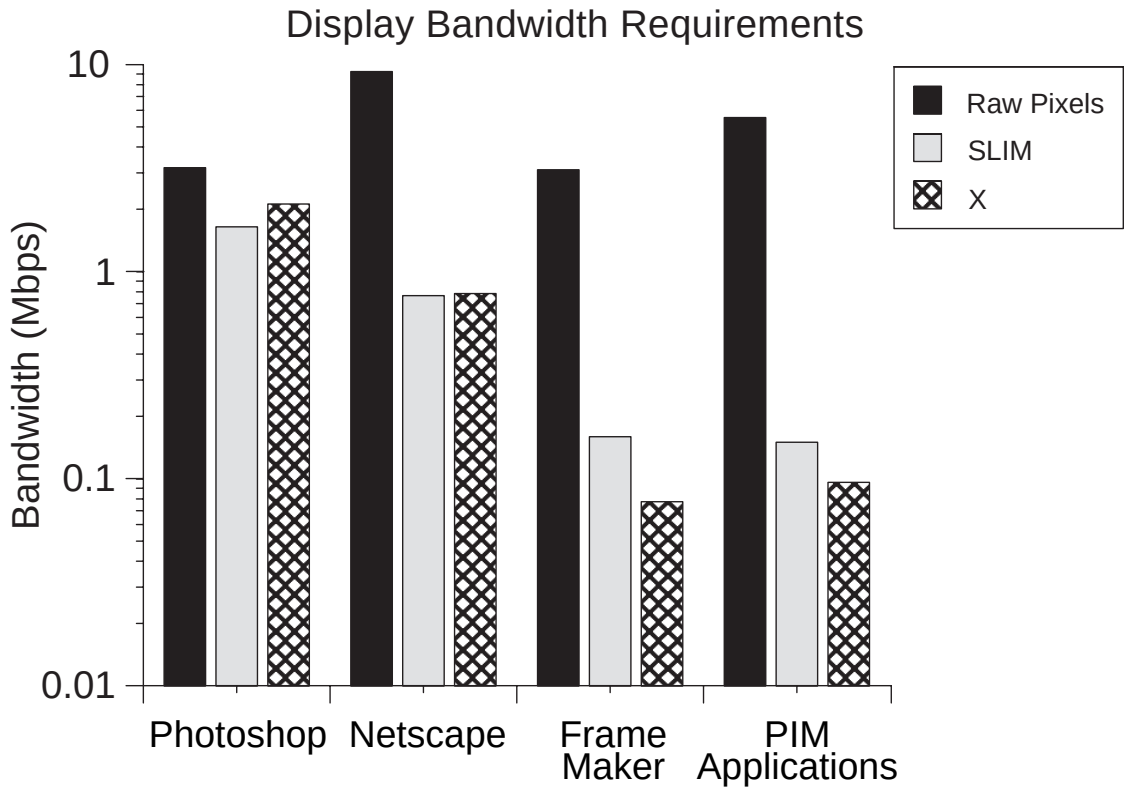


Figure 3-7 Average bandwidth consumed by the interactive benchmark applications under the X, SLIM, and raw pixel update protocols.

The second item to note is that the overall bandwidth requirements are quite small. We have already seen that network-induced delays do not adversely affect interactive performance, and this result demonstrates that network occupancy is also low. This implies that there is substantial opportunity to share the interconnection fabric, which we discuss below in Section 3.3.3.2.

3.3.2.7 Scalability of the SLIM Protocol

Although the SLIM protocol can comfortably accommodate the GUI-based benchmark applications with a low-latency, 100Mbps interconnection fabric, it is useful to consider how well it would operate over lower bandwidth connections. To obtain a sense of how interactive performance would be affected, we modified the X-server to restrict the bandwidth it could utilize, and then we ran our normal window sessions in the constrained setting.

At 10Mbps users could not distinguish any difference from operating at 100Mbps. To simulate a high-speed home connection, such as a cable modem or DSL, we tested the system with 1–2Mbps bandwidth limits. Performance was quite good, with only occasional hiccups when large regions had to be displayed. Finally, to simulate low-speed home connections such as ISDN or telephone modems, we tested the system with 56–128Kbps bandwidth limits. We found the performance to be extremely poor. It is possible to accomplish tasks such as reading e-mail or editing text, but interacting with GUI applications is slow and tedious. Of course, the SLIM protocol was not designed for such low-bandwidth connections, and optimizations like header compression, batching of command packets, and more aggressive data compression could have a dramatic effect.

To quantify this experience, we used the protocol logs from the resource profiles mentioned above and simulated transmitting the packets over lower bandwidth connections. We chose Netscape as a representative example and recorded the packet transmission delays in excess of the delays experienced at 100Mbps. Figure 3-8 depicts the cumulative distributions for a variety of bandwidth levels.

At 10Mbps the added packet delays are always less than 5ms, which is well below the 50–150ms threshold of human tolerance. The added packet delays at 1–2Mbps approach 50ms, entering the realm where humans would notice but consider acceptable. However, at 56–128Kbps we see a sharp increase in packet delays beyond 100ms, indicating that response time would be unacceptably high.

3.3.2.8 Server Memory Requirements

Physical memory is probably the most critical resource on the server after processor capacity. Thus, we analyze the memory requirements for each of our test applications, summarizing the results in Figure 3-9. For each of the GUI-based benchmark applications, we measured the resident set size (RSS) during the user studies. The RSS is the amount of physical memory an active process requires to avoid constant disk I/O due to paging. When given less memory than its RSS, an application will begin to “thrash,” constantly moving pages between memory and disk. Under these conditions, the application becomes

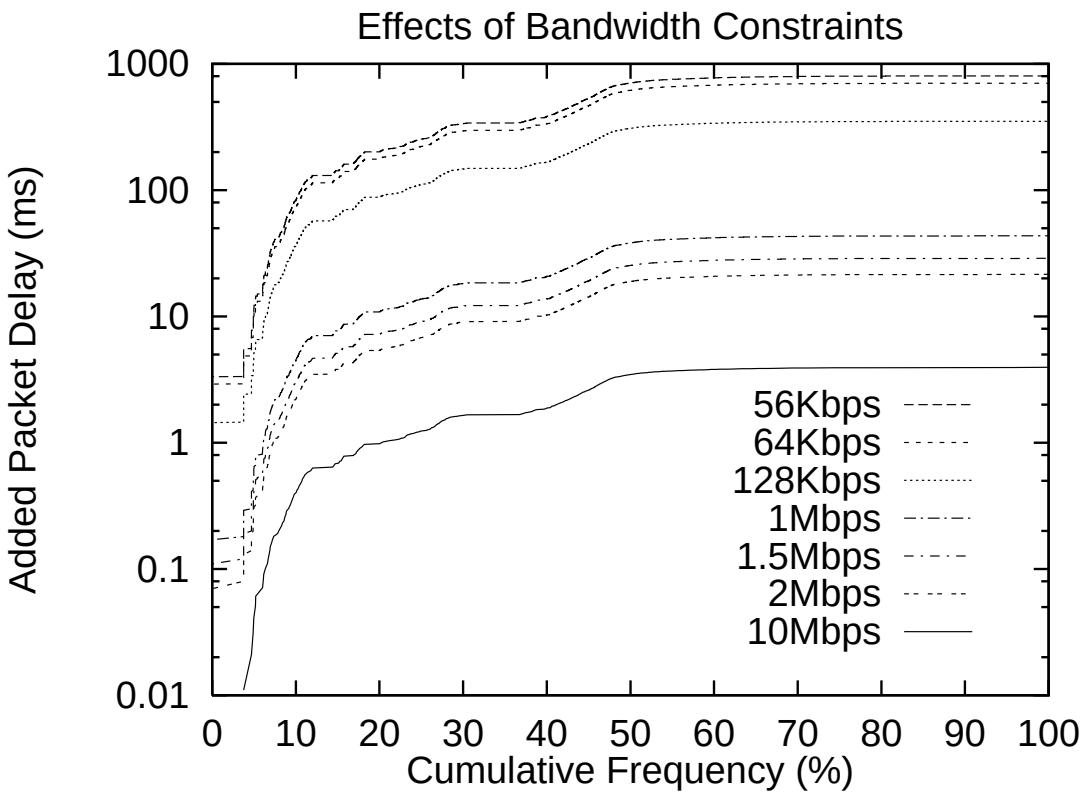


Figure 3-8 Cumulative distributions of added packet delays for Netscape. SLIM protocol commands were captured at 100Mbps and retransmitted on simulated networks with lower bandwidths. Bandwidth is averaged over 50ms intervals, and the histogram bucket size is 0.01ms.

completely I/O-bound and can only execute as fast as the disk allows. Performance is totally unacceptable in such situations. Thus, our results indicate the minimum memory requirements the server must meet in order to adequately support the test applications.

Since applications, can share memory (e.g. code segments), the cost of a shared page of memory need only be incurred once on a system, whereas the amount of required private memory increases linearly with the number of active instances. Thus, we present the RSS for single instances of the applications, with the understanding that only private memory requirements increase as more applications are run. In addition, the SLIM architecture is unique in that the X server is not run on the desktop, and it maintains its own copy of the frame buffer contents. Thus, we include the memory requirements an applications induces on its X server. Note, however, that each user requires only one X server per session. Thus,

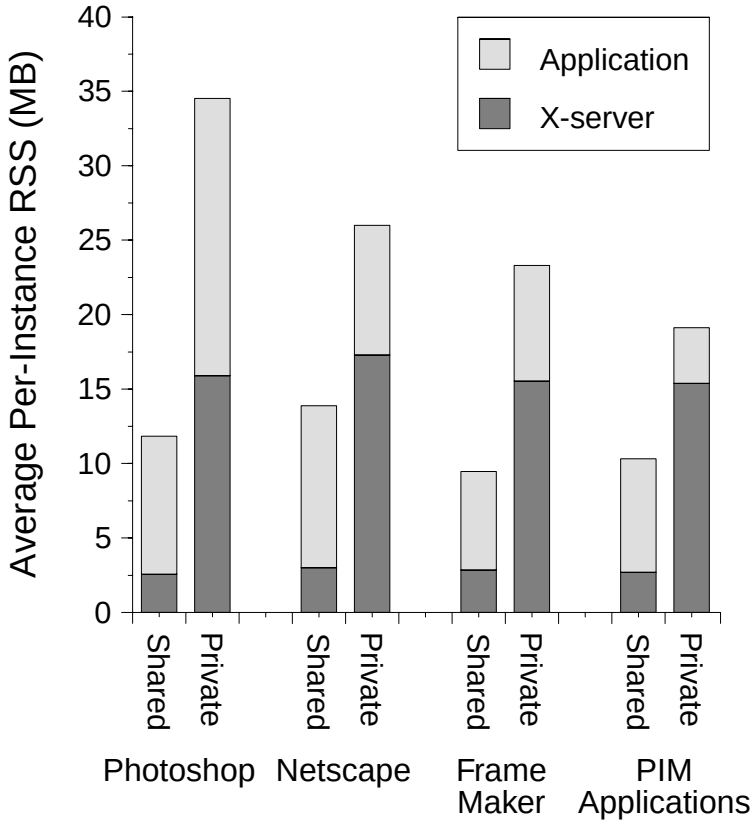


Figure 3-9 Server memory requirements for active applications.
Average resident set size (RSS) for a single instance of each test application are shown. We indicate memory that can be shared as well as private data. Because the X server does not run on the client, we include its requirements as part of the total cost.

the memory requirements of the X-server increase linearly with the number of users, not the number of active applications in the system.

From the chart we can see that memory requirements are fairly modest. Allocating 40MB of physical memory per user is sufficient to ensure adequate performance for any application, but there may be a paging cost when switching between applications. In addition, a pool of memory must be set aside to hold the shared pages for each of the applications that could be active in the system.

3.3.3 Interactive Performance of Multi-User Systems

The key advantage of SLIM and other thin-client architectures is the savings provided by sharing computational resources. In this section, we explore how interactive performance changes when there is interference from other simultaneously active users.

Many engineering workgroups have servers configured to support the requirements of their demanding computational activities, and we will continue to have these compute servers for executing long-running jobs in a SLIM system. Here, we are interested mainly in the sharing of machines for running those applications typically executed on our desktops.

3.3.3.1 Sharing the Server Processor

We want to determine how many active users a server can support while still providing good interactive performance. As discussed in Section 3.2, this is difficult to determine and is a subjective assessment that may be different for each individual. Because human perception is relatively slow, users can tolerate fairly large response times, i.e. 50–150ms. Thus, a processor can be oversubscribed but still provide good interactive performance as far as the users can tell. To account for this type of scenario, we must model the system in overload. However, large-scale user studies are impractical, and script-based techniques will fail due to timing constraints that cannot be met.

Experimental Approach

Our approach is to use a load generator to simulate active users. In addition to the traffic logs mentioned in Section 3.2, we supply the load generator with detailed per-process resource usage information, including CPU and memory utilization. These resource profiles were collected during the user studies with a tool that samples the number of CPU cycles consumed and physical memory occupied by each process at five-second intervals. This tool is similar to the one described in [58], and its overhead per analyzed process was measured and found to be insignificant. The load generator reads a resource usage profile and mimics its consumption of CPU, memory, network, disk and other resources over time. It does not replay the recorded X commands, SLIM commands, or other high-level operations. It merely utilizes the same quantity of resources in each time interval as the

original application did. In this way, we can produce the correct level of background load even when the resources are oversubscribed.

While the simulated user loads are running, we use an application with well-known properties and requirements as a yardstick to gauge interactive performance. This application repeatedly consumes 30ms of dedicated CPU time to simulate event processing, followed by 150ms of "think time." We defined the application in this way so that it would be more demanding than any other interactive application. In particular, it requires nearly 17% of the server, which is greater than the average CPU requirement for Photoshop (14%), Netscape (13%), Frame Maker (8%), and PIM (3%). Also, the interrupt rate is equivalent to a fast typist and is well beyond the sustained rates for any of our benchmarks. When the system becomes overutilized, the amount of elapsed wall-clock time needed to process the simulated event will exceed the required 30ms. We measure the value of this added delay as we increase the number of simulated active users.

Performance Measurements

The experimental setup is listed in Table 3-2, and the server had a single processor enabled. We modeled both CPU and memory loads for the active users, and the results are presented in Figure 3-10. To put them in perspective, we ran the GUI-based, benchmark applications ourselves under the experimental conditions and found that when the added delay on the yardstick application reached around 100ms, interactive performance of the overall system was noticeably poor, i.e. response time was sluggish for all activity, such as window manipulation, typing, and menu selection. As we can see from the average CPU requirements listed above, the processor on the server is significantly oversubscribed at this point. This is important because it demonstrates that a system can provide good performance for interactive applications even when the processor is fully utilized. In particular, we could tolerate up to about 10-12 simultaneously active Photoshop users, 12–14 Netscape users, 16–18 Frame Maker users, or 34–36 PIM users on the server (in addition to the yardstick application). Of course, other people will have different tolerance limits and application mixes. However, this experiment helps quantify how well a system

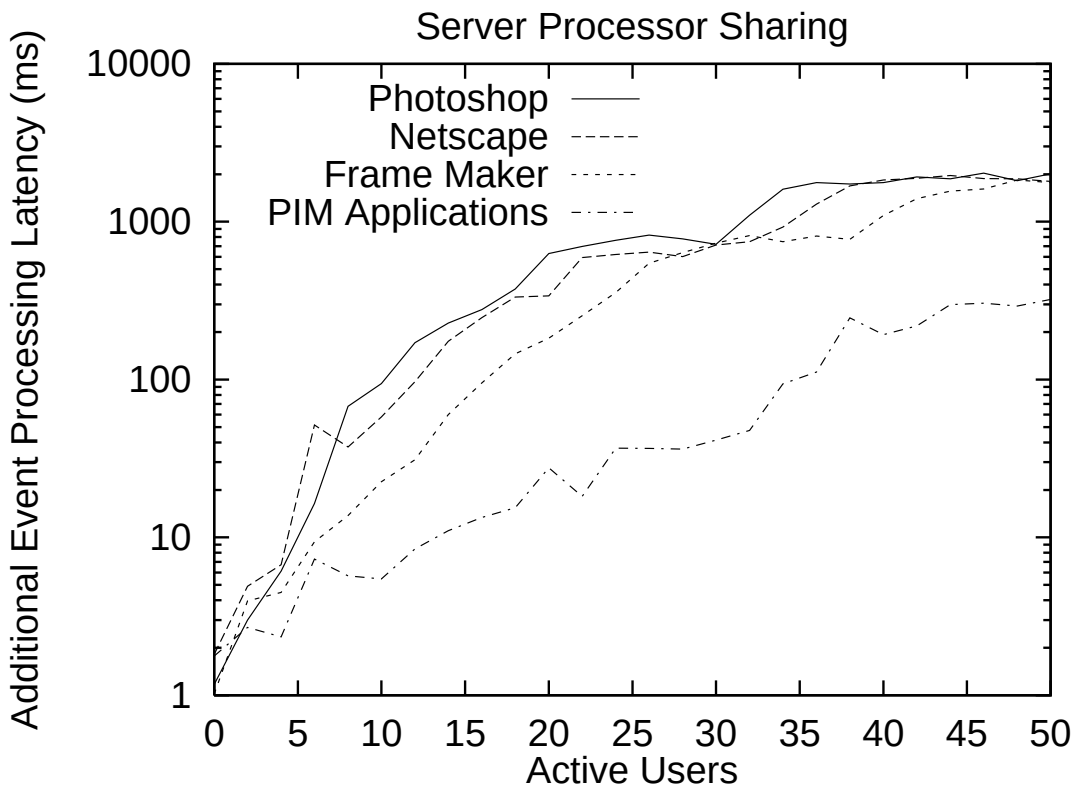


Figure 3-10 Response time on a shared system with a single processor. Average latency added to 30ms processing time for events occurring at 150ms intervals as the number of active users increases. There is one active CPU in the system, and user processor loads are generated with a trace-driven simulator which "plays back" previously recorded resource usage profiles.

will perform in a multi-user environment, and it provides a useful metric that is machine-independent.

3.3.3.2 Sharing the Interconnection Fabric

Although the preferred embodiment of the SLIM interconnection fabric is a dedicated private network, significant reductions in cost are possible by sharing bandwidth on the IF. Thus, we analyzed the cost of sharing the SLIM interconnection fabric.

Experimental Methodology

We again used the load generator discussed above to play back the network portion of the resource profiles to simulate active users on the IF. While the background traffic was being generated, we used an application with well-known properties as a yardstick to gauge

interactive performance. This application simulates a highly interactive user with sizeable display updates by repeatedly sending a 64B command packet to the server followed by a 1200B response and then 150ms of "think time." We measure the average round-trip packet delay as the number of active users is increased.

The experimental configuration is listed in Table 3-2. We used three workstations: one to act as a SLIM console, one to serve as a sink for background SLIM traffic, and one to act as a server. The machine which played the role of an active console executed the application described above and recorded round-trip packet delays for the network traffic. The machine acting as a server used the load generator to send background SLIM traffic to the sink host and responded to incoming packets from the simulated console. All workstations were connected to a network switch. In this way, the link to the server was shared by both the measured and background traffic, making it the point of contention in the system.

Experimental Results

Figure 3-11 presents the results for our benchmark applications. To put these findings into perspective, we again ran the benchmark applications for ourselves under the experimental conditions. We found the system to be quite usable until packet delays for the test application hit about 30ms, at which point response time suffered greatly and packet loss became a problem. Thus, we could tolerate up to about 130–140 simultaneously active Photoshop or Netscape users, or about 400–450 Frame Maker or PIM users on a shared network. This is interesting because it demonstrates that the network is a less critical resource than the processor, memory, or swap space on the server. In particular, a 100Mbps Ethernet link can support an order of magnitude more users than a 300MHz UltraSPARC-II processor.

3.3.3.3 Case Studies

Since the Sun Ray 1 is a commercial product, we have had the opportunity to monitor its use in real-world settings. In this section we present two case studies of the SLIM architecture during actual operation.

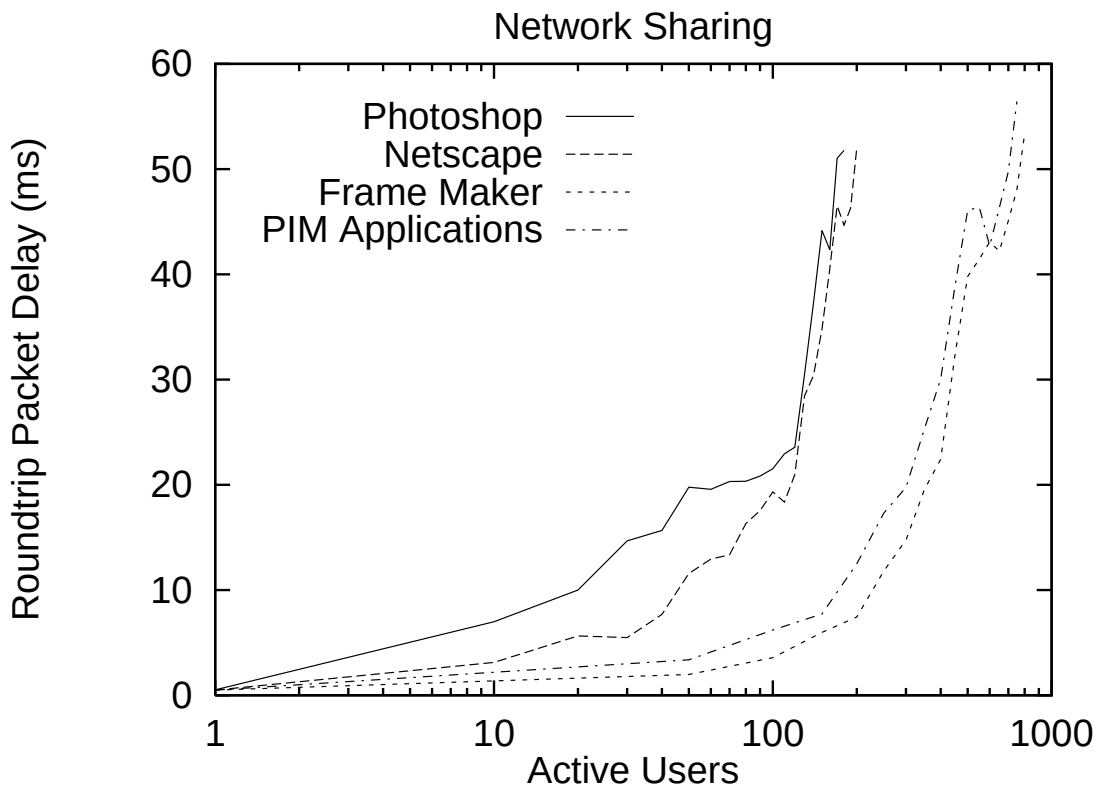
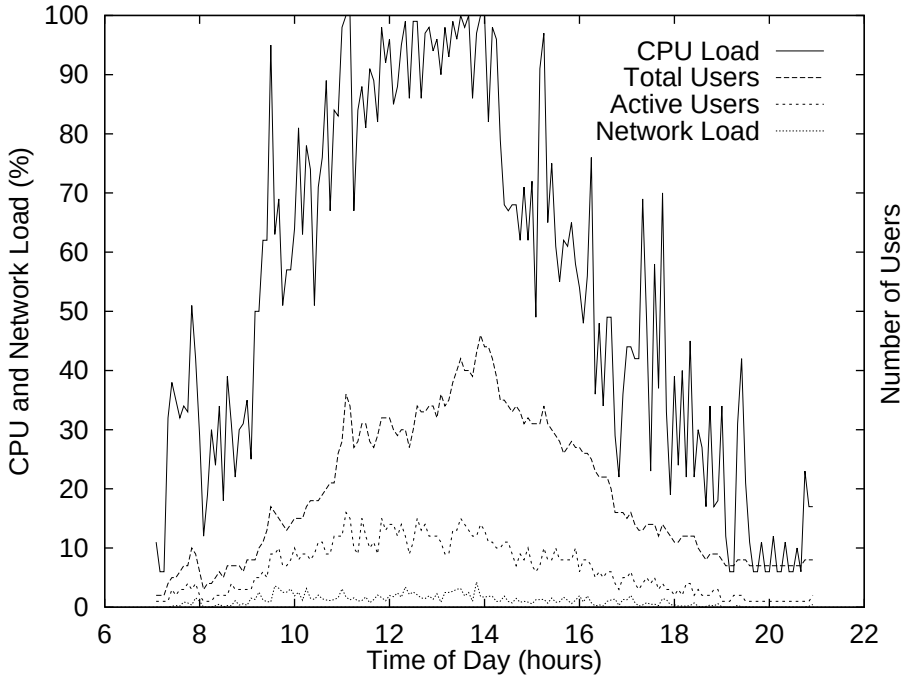


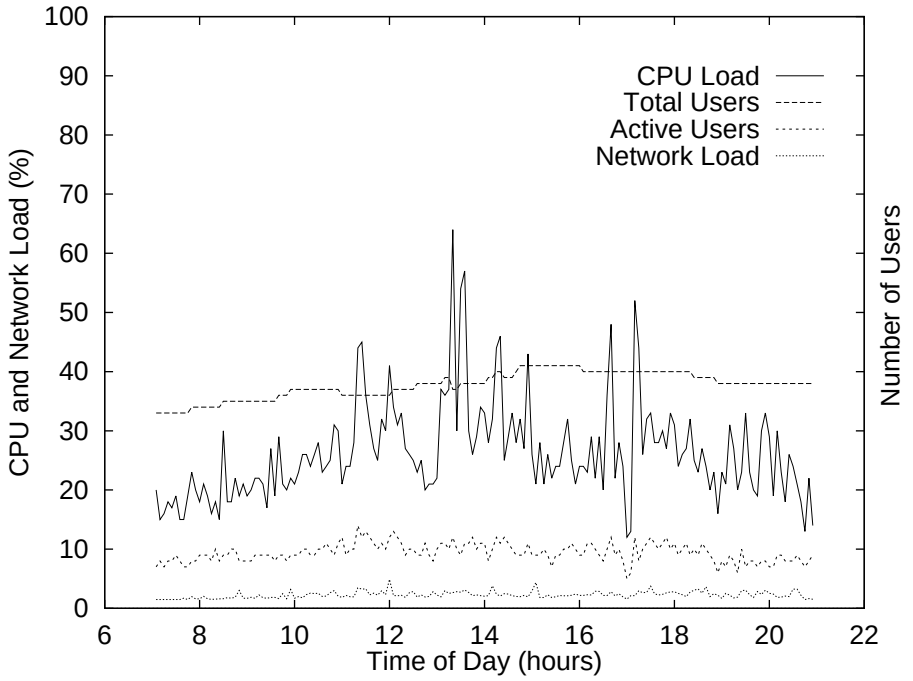
Figure 3-11 Latency on a shared network.
Average round-trip delay for 64B upstream packet followed by 1200B downstream packet. User traffic loads are generated with a trace-driven simulator which "plays back" previously recorded resource usage profiles.

At two installations we used standard resource monitoring tools (e.g. ps, netstat, vmstat) to collect performance data continuously over the course of several days. Snapshots of aggregate CPU load, aggregate network bandwidth usage, and number of active users were taken every 10 seconds throughout the day. For each site, Figure 3-12 presents the day-long profile which has the highest average processor load and the highest number of active users, and we report the maximum value recorded in each five-minute period.

The first graph presents a load profile from a university lab that was previously supported by a collection of X Terminals. Students work on programming assignments and other normal computing tasks, and major applications include MatLab, StarOffice, Netscape, e-mail and compilers. The server is a Sun Enterprise E250 with two 400MHz UltraSPARC-II CPUs, 2GB of RAM, a 1Gbps link to the IF, 13GB of swap space, and the Solaris 2.7 operating system. There are 50 terminals attached to the server, and at the



a) University Lab



b) Product Development Team

Figure 3-12 Day-long plot of aggregate CPU utilization, network bandwidth, and active users for real-world installations of the Sun Ray 1 system.

busiest part of the day many are in use (Total Users in the graph). However, far fewer users are actively running jobs on the system (Active Users in the graph). Both processors reach full utilization during peak periods. However, aggregate network load is below 5Mbps, making the 1Gbps uplink massive overkill.

The second graph presents a load profile from our computer product development team. Engineers and other staff use CAD tools for hardware design, text editors and compilers for software development, as well as Netscape, StarOffice, Citrix MetaFrame, e-mail, calendar, etc. for office productivity. The server is a Sun Enterprise E4500 with eight 336MHz UltraSPARC-II CPUs, 6GB of RAM, a 1Gbps link to the IF, 13GB of swap space, and the Solaris 2.7 operating system. Across two buildings, there are over 100 terminals attached to the server, and many are in constant use (with a smaller fraction in active use). Users in this group leave their sessions active and frequently utilize the mobility feature via the smart card. Again, aggregate network load is below 5Mbps. The server processors are never fully occupied, and there is certainly sufficient headroom to scale down the server. A major benefit of this system is the reduction of administration cost from the more than 100 workstations previously used by this group to the single, shared server.

3.3.4 Supporting Multimedia Applications

Real-time applications, such as video or 3D-rendered games, represent an important class of applications that place high demands on system resources to provide desired performance. As such, these kinds of applications have traditionally been viewed as totally unsuitable for remote display systems. In this chapter, we investigate the ability of SLIM to support such applications.

3.3.4.1 Multimedia Application Requirements

Multimedia applications differ from the ones studied previously because they have updates that occur at high rates, affect many pixels, and are not tied to human interaction. Large amounts of pixel data are constantly streaming to the desktop, and so bandwidth and processing demands are expected to be prohibitive. Thus, these applications represent the worst-case scenario for a SLIM system.

In addition, multimedia applications typically render directly to the frame buffer to maximize performance. To achieve the same effect in a SLIM system, the application must utilize the SLIM protocol directly. This means that there is a porting cost and implementation overhead, and as we will see below, the added overhead can be quite high. However, it is fair to point out that reducing the resolution of the media streams and scaling them locally on the SLIM console reduces this extra cost to a manageable level, thereby enabling applications to achieve display rates well beyond human tolerance limits with little or no noticeable degradation.

Because SLIM desktop units have so little resources, we must ensure that multimedia applications do not starve other applications for protocol processing service on the console. Therefore, we implemented a simple network bandwidth allocation mechanism on the Sun Ray 1 console. Under this scheme, applications (e.g. the X-server or Quake) executing on possibly different servers make bandwidth requests to the display console based on their past needs. These requests are transparent to the application programmer, as they are made by the X-server on behalf of X applications and by the video library (see Section 2.1.2) on behalf of multimedia programs. The console sorts the requests in ascending order and grants them one at a time until a request exceeds the available bandwidth, at which point all remaining requests are granted a fair share of the unallocated bandwidth. In this way, high-demand multimedia applications can run while other traditional applications still receive good interactive service from the console.

3.3.4.2 Experimental Set-up

To analyze the suitability of SLIM for these applications, we experimented with three multimedia applications: an MPEG-II player, an NTSC video player, and Quake, an interactive 3-D game from id Software. Using the software library mentioned in Section 3.1.3, we enhanced these applications with the ability to display directly on Sun Ray 1 consoles via the SLIM protocol. The applications transmit synchronized audio, and they have similar real-time quality of service demands. More specifically, humans require a frame rate of at least 24Hz to achieve smooth display and proper motion rendition. Quake has the additional requirement of timely user interaction.

Although Quake is a 3-D game, it does not use standard 3-D interfaces, such as OpenGL. Thus, we use it more to assess interactive game performance than to analyze rendering performance for 3-D graphics. Since Sun Ray 1 consoles have no hardware acceleration for 3-D operations, 3-D graphics performance is directly related to the speed of the server processor anyway.

Using the test configuration listed in Table 3-2, we demonstrate that despite substantial resource requirements, even high-demand applications are supported by the SLIM architecture. In particular, the bandwidth and processing capabilities of the consoles are more than sufficient to meet the demands of these applications, and it turns out that server performance is the primary bottleneck. Of course, it is important to note that in all these cases, a 10Mbps (or lower) interconnection fabric would not be able to provide adequate performance.

3.3.4.3 Playback of Stored Video

Support was added to the Sun Microsystems ShowMeTV video player to utilize the SLIM protocol, and we use it for stored video playback. In this experiment, we selected an MPEG-II clip with a resolution of 720x480 and used the CSCS command with 6 bits/pixel compression. We extract frames from the ShowMeTV MPEG-II codec at the point where they are in YUV format, and the SLIM video library is then used to transmit them to the desktop unit.

This application nearly consumes an entire CPU, and disk I/O and video decompression on the server are the performance-limiting factors. The displayed frame rate was fairly uniform at 20Hz (roughly 40Mbps), which is quite good, given the resources available on the desktop. Still, full frame rate (30Hz for this clip) can be achieved by sending every other line and scaling at the desktop. Degradation is not noticeable for the most part, and the bandwidth would be reduced by half.

3.3.4.4 Playback of Live Video

To test live video, we used a custom application that acquires JPEG-compressed NTSC fields from a video capture card, decompresses them up to the point where YUV data are

available, and transmits them to the desktop via the CSCS command. Since NTSC is interlaced, we can capture only 640x240 fields and scale up to full size (640x480).

The decompression operation fully consumes the processor, and this application is not multi-threaded. Thus, the server CPU is the performance-limiting factor, and the displayed frame rate ranges from 16Hz to 20Hz (roughly 19–23Mbps), depending on characteristics of the video. To create a situation in which protocol processing on the console is the bottleneck, we simulate 4-way application-level parallelism by simultaneously executing four half-size (320x240) players, which is equivalent to running one parallel version of a full-size player. In this case, we observe a display frame rate of 25–28Hz (59–66Mbps). Thus, if we were to multi-thread this application, it would display full-size video that is quite watchable (at the cost of four heavily-utilized CPUs).

3.3.4.5 3D-Rendered Video Games

We use Quake as a representative example of 3D-rendered games. Running it under X produces poor results, and the key to increasing performance would be to use the YUV color space directly in the application. However, we only had access to the code that puts pixels on the display, and it would be a costly porting operation anyway. So, the next best option is to add a translation layer that converts frames to a format suitable for use by the SLIM CSCS protocol command.

When the game engine renders a frame, it produces 8-bit, indexed-color pixels. Our translator calculates a YUV lookup table based on the RGB colormap. To display a frame, we convert the 8-bit pixel values to 5-bit YUV data via table lookup and color component subsampling. Then, the frame is transmitted to a Sun Ray 1 console.

When we run Quake at the full-size resolution of 640x480, the display rate ranges from 18Hz to 21Hz (roughly 22–26Mbp), depending on scene complexity. Although this is good performance, we found the interactive experience to be somewhat lacking. However, if we run Quake at 3/4 resolution (480x360), display rate improves to 28–34Hz (roughly 20–24Mbps), which we found to be playable despite occasional hiccups. The performance-limiting factor in these cases was the CPU, due almost exclusively to the high

cost of translation. For example, at the 640x480 resolution it took roughly 30ms/frame to do the YUV conversion and 13ms/frame to transmit the data, which means that the upper bound on display rate was only about 23Hz.

To further improve performance, we could parallelize the application. Because we had limited access to source code, we simulated the effect of parallelizing an instance of the game at full resolution (640x480) by running four instances at 320x240 resolution, thereby creating a situation in which the limiting factor was the protocol processing performance of the desktop unit. With this setup we achieved frame rates ranging from 37Hz to 40Hz (roughly 46–50Mbps), which we found to be smooth and responsive with no hiccups or other noticeable effects.

3.4 Summary

In this chapter, we have presented the implementation of the Sun Ray implementation of a SLIM system, as well as an evaluation of its interactive performance. Our results demonstrate the feasibility of this architecture to support the decoupled human interface model we propose in our remote computational service architecture.

SLIM consoles are intended as replacements for desktop machines, but because they are simple access devices with no computational resources to speak of, we are concerned with the user experience and not standard performance metrics. Because little has been done in the way of evaluating the interactive performance of a general system, part of our contribution is a methodology for analyzing interactive systems. Our findings are also useful when analyzing the interactive performance of other systems.

Our methodology is to base the experiments on modern, highly-interactive applications such as Photoshop and Netscape, as well as streaming video and 3-D games. We measured these applications along dimensions that govern their interactive performance, such as input rates, display update rates, and bandwidth requirements. Since interactive jobs require actual users to provide input, we conducted a set of user studies to collect application profiles. As such experiments are expensive to run, we logged all the information related to their network traffic and resource utilization. In this way, we can

investigate different aspects of the system by post-processing the data, rather than conducting more user studies. The data collection requires minimal resources, and thus does not perturb the results.

A difficult problem we had to address was how to measure the effect of sharing on interactive performance and to determine the level of sharing that can be supported by a system. Our solution is to utilize a yardstick application, i.e. one with well-defined characteristics. We quantify the effect sharing has on interactive performance by measuring the additional latency experienced by the yardstick application under different loads.

Using this methodology, we evaluated the performance of the Sun Ray 1 implementation and compared it with the X protocol. Our results demonstrate that the SLIM protocol is capable of supporting common, GUI-based applications with no performance degradation. Surprisingly, the low-level SLIM protocol does not translate to a greater bandwidth demand when compared to X. This is because X was found to only optimize applications with low-end bandwidth requirements. A Sun Ray 1 console can display a 720x480 video clip at 20Hz and allows Quake to be played at a resolution of 480x360. The server, and not the bandwidth to the console, turns out to be the bottleneck for these applications. Finally, our experiments show a high potential for resource sharing. Depending on the application class, anywhere from 10 to 36 active users can share a 300MHz processor without any noticeable degradation of interactive performance. While the network is often regarded as the major performance bottleneck in thin-client systems, it can support more active users (by an order of magnitude) than the processor and memory. Our results also indicate that service to the home over broadband connections would have acceptable performance, but this approach would not work well in high-latency, low-bandwidth settings such as dial-up connections to the internet.

4 Compute Capsule Implementation and Experience

To demonstrate the feasibility of our remote computational service architecture in a real-world setting, we implemented compute capsules within the context of the Solaris Operating Environment, which is the most widely used version of the Unix operating system today. Our implementation is based on Solaris version 2.7, which was the latest release when we began working on the prototype. In this chapter, we present the details of our compute capsule implementation, and we report on our experience using the prototype system.

When we started building compute capsules, we considered both the Solaris and Linux kernels for the basis of our implementation. Many modern researchers and developers have chosen to use Linux because its source code is freely available, and the changes could make it back into the main distribution so they become available to the growing Linux community. Linux also has the advantage of having a relatively small code base, making it fairly easy to modify. Solaris, on the other hand, is a large, monolithic kernel with ill-defined and poorly documented interfaces between its various modules. In the end, we decided to use Solaris because we had full access to the source code, and our prior experience with its internal workings and development tools made it the more appealing choice. Still, this decision came at the price of significant implementation effort due to the size and complexity of the Solaris kernel.

Although many systems have been built that support checkpoint/restart or process migration, no system to date has been able to demonstrate the suspension, migration, and resumption of an arbitrary collection of unmodified, communicating, graphical applications (e.g. as is the case for a typical desktop login session). This chapter describes

our experience with compute capsules for exactly this type of scenario. In addition, we discuss the overhead that supporting capsules introduces into the system, as well as the performance of capsule-specific operations, such as suspension and revival of capsules. Finally, we discuss how capsules could be used in a large-scale, real-world setting.

4.1 Compute Capsule Implementation

We implemented compute capsules by modifying the operating system, and this choice is based on the following three factors. First, we want existing applications to run without modification, re-compilation, or re-linking. Second, while we prefer to operate in user space as much as possible, kernel-level control is often required. For example, certain functionality, such as namespace management, is sensitive and must be carefully controlled to avoid security breaches. Because the trusted computing base cannot safely include things like user-level libraries, such features must be implemented in the kernel. Also, maintaining a virtual process namespace requires the knowledge of when processes exit, but not all processes make the appropriate system call, e.g. if they receive a signal and abort. Third, the kernel caches and maintains a great deal of state that is not accessible to user-level code. An implementation in the kernel has direct access to this state, whereas an implementation outside the operating system would have to duplicate the state management performed by the kernel. By implementing capsule functionality within the kernel, existing applications can run unchanged, and we avoid problems associated with security and identifying capsule state.

We have implemented compute capsules as a set of extensions to the Sun Solaris 7 Operating Environment. It is a thin layer of software that we have interposed between the application layer and the kernel, and our system consists of nearly 3700 lines of C code. This includes a set of modifications to the kernel code, some new functions in the system libraries, and a set of new utility programs.The decomposition of source code functionality is listed in Table 4-1.

Code Location	Functionality	Lines of Code
Utilities	Capsule management (create, ACLs, checkpoint, restart, etc.)	328
	Modified session login	29
	Modified remote access (rlogin, rsh, ftp, telnet, rexec, rcp)	14
Library	Create/join capsules, file system view management	249
	Checkpoint/restart signal handlers	99
Device Drivers	Keyboard, mouse, graphics controller	118
Kernel	Checkpoint/restart signal handling	42
	Create/join capsules	192
	Namespace translation	282
	ACL management and capsule management system calls	135
	Maintaining file pathnames	88
	Capsule checkpoint	1093
	Capsule restart	1008
Total		**3677**

Table 4-1 Size of C source code for compute capsule implementation. Only lines containing semicolons were counted, which reflects the number of actual instructions fairly well.

4.1.1 Capsule Management

Our implementation provides support for capsule management through the `capsule_create`, `capsule_join`, and `capsule_acl` system calls (see Table 2-2), as well as an interface to the underlying operating system. We discuss these below.

4.1.1.1 Capsule Directory Service

Our prototype implementation is used in a small, workgroup environment with shared file servers. In this setting, the simplest means of building the capsule directory service was to

create a database with the necessary capsule information. This includes a logical name for the capsule, the name and IP address of its current host, and its status. The status is indicated implicitly by the location, i.e. a host name indicates that it is currently running, whereas a file pathname for the state information indicates that it is suspended on disk. This simple database is mapped into each capsule at a well-known location (see Section 4.1.2.1).

4.1.1.2 Creating and Joining Capsules

To ensure proper use, the `capsule_create` and `capsule_join` system calls are privileged operations that may only be invoked by the root user. We incorporated these routines into the login facility and remote access utilities (i.e. login, telnet, rsh, rlogin, ftp, rcp, rexec), thereby allowing users to create and join capsules transparently. These utilities make the `capsule_create` or `capsule_join` system call, create a new process (such as a shell or file transfer interface) belonging to the owner of the new capsule, and then they exit immediately. In addition, we added a new system program that allows a user to create a new capsule to execute a specified application. This is useful for creating capsules with arbitrary contents, not just ones that represent login sessions.

Capsule creation is implemented partly in the kernel and partly in user space within the standard system library (which represents the Solaris application binary interface). First, within the kernel, our implementation checks the user's permissions, allocates a compute capsule structure, initializes the name translation tables, assigns a unique name to the capsule, and moves the invoking process into the new capsule. At this point, the parent process resides in a different capsule, which creates a non-conforming means of inter-capsule communication (e.g. through signal or wait system calls). Thus, we restrict top-level capsule members to be children of the Unix init process, which is responsible for waiting on orphaned processes to exit. Because init is guaranteed to exist on every Unix system, it does not add a host dependency. Once kernel processing is complete, control returns to the library interface code, which registers the capsule name in a well-known database service that is mapped into each capsule, i.e. the capsule directory service. Finally, the file system view is established (see Section 4.1.2.1), and the invoking process resumes user-level execution as the sole member of the new capsule.

To join a capsule, the user looks up its logical name in the capsule directory to obtain its current location and then issues a `capsule_join` on the target host. The system checks the user's permission and then moves the invoking process into the destination capsule. The process then severs its ties to its former capsule and becomes a child of init.

As mentioned above, capsule membership is inherited by the descendants of a process. To accomplish this goal, we modified the process creation system calls, i.e. fork, vfork, etc. Once the child process is created, the kernel adds it to the same capsule as its parent. Similarly, we modified the kernel routine which reaps a defunct process so that it also removes the exiting process from its capsule. When the capsule contains no processes, it is removed from the system: kernel resources are released, the file system view is unmapped, and the capsule name directory is updated.

4.1.1.3 Access Control

Each capsule includes an access control list (ACL), which identifies users who are authorized to join the capsule. The `capsule_join` system call first checks that the requestor is on the ACL before moving a process into the target capsule. Users not on the list cannot add processes to the capsule; this restriction even applies to administrative users. Thus, although a setuid program may assign root-level access rights, it will not enable a process to illicitly access other capsules. The capsule ACL is initially empty, i.e. only the owner may join. We implemented a new system call (`capsule_acl`) for modifying the ACL, and only the capsule owner is authorized to invoke it. Users may be added and deleted, and the entire ACL may be disabled altogether. In addition, we created a utility program to simplify ACL manipulation.

4.1.1.4 Access to the Underlying Operating System

In our implementation, the underlying operating system is accessed by logging into a machine as the user root. The user enters the administration capsule, which has the same effect as logging into a traditional machine as root, e.g. all processes and machine-local PIDs are visible. Because capsule membership is inherited, switching user identities (su) or running a setuid program within a capsule will not have the same effect. To provide an interface identical to the traditional operating environment, our implementation must avoid

normal capsule-related operations, such as name translation. Thus, wherever a capsule operation occurs, our system first verifies that the active process is within a standard capsule before proceeding.

4.1.2 Capsule Naming

The member processes of a compute capsule can access two classes of objects: those referenced through the file system, and those referenced through the standard system call interface. We consider the naming of each class in the following two sections.

4.1.2.1 File System View

To create the file system view for a compute capsule, we leverage heavily from existing technology. Each user is assigned globally accessible capsule storage to contain the view. We use NFS [57] to provide global storage, and this requires that each machine in the system have uniform mount hierarchies and permissions, which may not scale globally. However, it is sufficient for a prototype in our workgroup settings, and a larger scale file system, such as AFS [25], or a web-based file system [60] could just as easily be used in its place.

As mentioned in Section 4.1.1.2 above, once a capsule is created by the kernel, user-level library code completes the establishment of a file system view. Elements from the underlying file system are mapped into the view based on the contents of two configuration files. The first file provides site-wide defaults, which enable all capsules to access standard software and data in the system. The second file is located within a user's capsule storage and contains custom mappings.

The default set of file objects that are mapped into a file system view include the standard collection of directories necessary for the normal operation of any system. These include /bin for applications, /lib for system libraries, /usr/man for manual pages, etc. These system files are typically cached on local disks, and we assume their contents are uniform across machines in the system. Given that such files are rarely changed and that system administrators maintain consistency among the machines in the system, this is a reasonable

assumption. The special directory /capsule is mapped into all capsules, and it contains global capsule information, such as the capsule directory service database.

To create the file system view from the configuration files, mount points are established for each of the required directories, and then the loopback file system is used to map them into the view from the local machine. Although capsule owners are free to modify their file system views, we do not want them to modify the underlying local system-level files (for which they may not have permission). Thus, mounts of directories such as /lib are made read-only. In our implementation, the copy-on-write feature is functional but a bit crude. If users wish to modify a directory such as /lib, they make a copy in their personal view and remove the map to the underlying file system. Then, changes only affect the copy, which will remain in the user's private capsule storage. For directories like /tmp, the system creates a unique subdirectory within it and then uses the loopback file system to map the subdirectory into the view with the original directory name. The contents of such directories are cached on the local machine but maintained in capsule storage. This ensures that the proper semantics for files in the /tmp directory are maintained, i.e. they are on local storage and available from a standard location. Once mounts have been established, the chroot system call re-assigns the root of the file system to be the root of the file system view, which gives capsules the illusion of a private file system.

4.1.2.2 Resource Naming

For each capsule within the system, our implementation provides a private namespace, which is comprised of two components: the file system view outlined above, and the virtualization of resource names, which we describe here. Each resource object referenced by a capsule is assigned a virtual name that is unique within the capsule. When a process passes a reference to a resource object to the kernel, the virtual name is mapped to the physical resource in the local system. This procedure is reversed for return values from the kernel.

We accomplish this by adding two name translation tables to the kernel. One table maps virtual names to physical resources, and the other performs the reverse mapping. They are implemented as hash tables that are segmented based on capsule identity, i.e. a logical table

for each capsule. A hash table lookup is performed at all points within the system where interface objects are named, including system entry points (e.g. system calls), special files (e.g. the /proc process file system), and ioctl calls. As new system objects are created, they are assigned capsule-local names and entered into the hash tables for future lookups. Similarly, when a resource is no longer available, its virtual name and hash table slot are freed for later re-use. References to objects that are not in the tables return an error. For example, the ps command and listing the contents of /proc only return PIDs for capsule members.

One goal of virtualization is to provide host-independence, and so objects with host-specific names require translation, including the following: process identifiers (PIDs); Unix System V shared memory segment identifiers (SHMIDs), message queue keys, and semaphore keys; pseudo-terminal device names; and internet addresses. Our implementation includes support for PID and pseudo-terminal name translation. Although messages, semaphores, and shared memory are simple to virtualize, we omitted them in the interests of rapid prototyping. Although virtualizing internet sockets is fairly straightforward (see Section 2.3.4), it requires substantial implementation effort and infrastructure and is therefore outside the scope of this research.

Process name translation is perhaps the most complex mapping to perform because it is so prevalent throughout the system interface. So, we will use it as a representative example to further detail our implementation. We maintain a hash table that is indexed by triples consisting of the user identifier for a capsule owner, the instance number of the capsule, and the PID. The value returned is a virtual PID, and a reverse lookup table is used for the opposite translation. Wherever PIDs are passed between the kernel and user-level code, we inserted the translation. In addition, operations that request PIDs from outside the capsule return an error. Thus, commands like the ps utility can only list the processes within its capsule, thereby giving the illusion of a private machine. In addition to the user-kernel interface, we also modified the special process file system /proc to translate and restrict PIDs. Whereas listing the contents of /proc would normally display the system PIDs for all processes on the local machine, under our implementation only virtual PIDs in the current capsule are accessible.

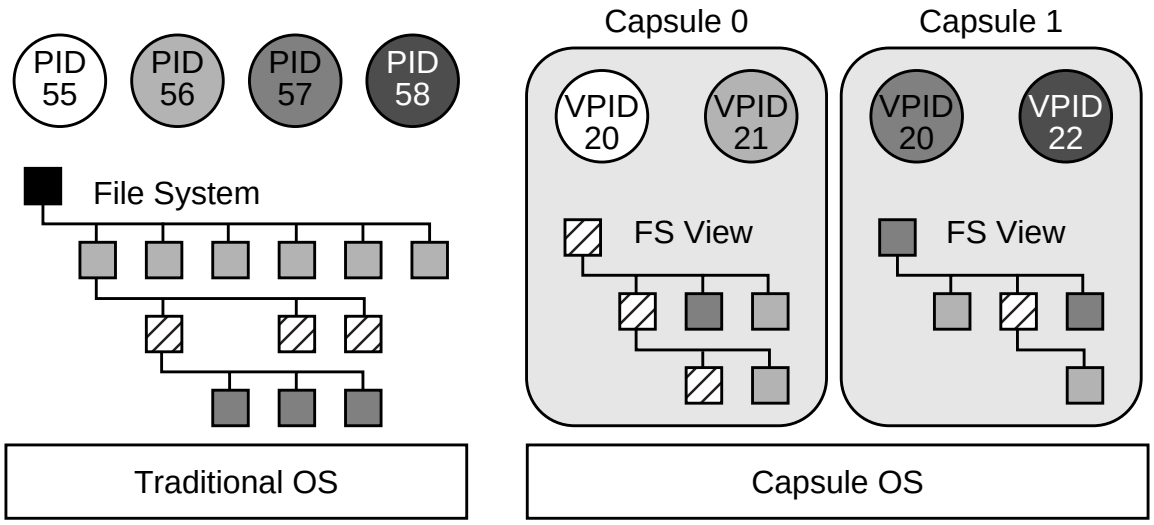


Figure 4-1 Example of naming with compute capsules.

4.1.2.3 Example Scenario

To help solidify the mechanisms for capsule naming, consider the example system depicted in Figure 4-1. Two views of the same system are presented. When accessing the underlying traditional operating system, we see four processes (numbered with PIDs 55 through 58) and a file system hierarchy that may include remote mounts. From the perspective of the capsule system, there are two capsules, each containing two processes and separate views of the file system. Consider process 55, which has been assigned VPID 20 in capsule 0. If this process were to issue the ps command or execute 'ls /proc', the result would be to return the values 20 and 21. If it were to execute the shell command 'kill 56' or 'kill 22', it would receive an error since 56 is the PID for the process with VPID 21, and there is no process in capsule 0 with VPID 22. Similarly, if it tried to issue a cd command to the directory indicated by a black square, it would receive an error.

4.1.3 Persistence and Relocation

Compute capsules are self-contained active computing environments, i.e. the complete state of the computation and its environment is included within the capsule. In this section, we outline the components of the state and describe how our implementation employs it for capsule persistence and relocation.

4.1.3.1 Re-partitioning State Ownership

As shown previously in Figure 2-1, we re-partition ownership of computing environment state between capsules and the operating system. Capsule state includes data that are host-specific, cached on the local machine to which the capsule is bound, or not otherwise globally accessible. This includes the following information.

- *Capsule State*: Name translation tables, access control list, owner, name, etc.
- *Processes*: Tree structure, process control block, machine context, thread contexts, scheduling parameters, etc.
- *Address Space Contents*: Because they are available in the file system, contents of read-only files mapped into the address space (e.g. the application binary and libraries) are not included unless explicitly requested. All other contents are included, including shared pages between different processes. (Recall that shared memory between capsules is a non-conforming means of inter-capsule communication and is thus disallowed.)
- *Open File State*: Only file names, permissions, offsets, etc. are required for objects available in the global file system. However, the contents of personal files in local storage (e.g. /tmp) must be included. Because the pathname of a file is discarded after it is opened, for each process we maintain a hash table that maps file descriptors to their corresponding pathnames. In addition, some open files have no pathname, i.e. if an unlink operation has been performed. This is a fairly common technique applications employ to ensure that a file will be automatically discarded upon termination, and so we include the contents of such files in the capsule as well.
- *IPC Channels*: Inter-process communication state has been problematic in most prior systems. In our implementation, we added a new interface to the kernel modules for each form of IPC. This interface includes two complementary elements: export current state, and import state to re-create channel. For example, we modified the pipe/fifo module to export the list of processes attached to a

pipe, its current mode, the list of streams modules pushed onto it, file system mount points, and in-flight data. When given this state data, the system can re-establish an identical pipe.

- *Open Devices*: By adding a state import/export interface similar to that used for IPC, our implementation supports the most commonly used devices: keyboard, mouse, graphics controller, and pseudo-terminals. The mouse and keyboard have very little state, mostly the location of the cursor and the state of the LEDs (e.g. caps lock). The graphics controller is more complex. The video mode (e.g. resolution and refresh rate) and the contents of the frame buffer must be recorded, along with any color tables or other specialized hardware settings. Supporting migration between machines with different graphics controllers is troublesome, but using the standard remote display interface of the SLIM architecture addresses that issue. Further, the SLIM architecture supports disconnected devices transparently, thereby eliminating the need to provide special migration support for all possible devices. Pseudo-terminal state includes the controlling process, numerous control settings, a list of streams modules that have been pushed onto it, and any unprocessed data.

Capsules do not include shared resources or the state necessary to manage them (e.g. the processor scheduler, page tables), state for kernel optimizations (e.g. disk caches), local file system, physical resources (e.g. the network), etc.

4.1.3.2 Capsule Persistence

Given this ability to fully obtain the state of a capsule, suspending it in persistent storage is fairly straightforward. We have implemented this functionality with a new system call (`capsule_checkpoint`). To checkpoint a capsule, one of its member processes belonging to the capsule owner issues this call, and the system allows only one outstanding request. A privileged administrator may also suspend a capsule from the administration capsule. This operation begins with a trap into the kernel, and then the system sends a signal to all processes within the capsule, notifying them that a checkpoint operation is about to take place. This signal forces processes to trap into the kernel and enter the idle state. Once

all processes are halted, the capsule state will be totally quiescent, at which point it is safe for kernel threads to record the capsule state. Then, the capsule resumes execution or exits the system.

Applications can catch the checkpoint signal and perform whatever actions are desired, but they cannot ignore it, because then the process would not execute the checkpoint operation inside the kernel. In addition, the handler can execute arbitrary code and may never return. Similarly, a process may be executing within the kernel and not notice the signal. Thus, we added a time-out on the checkpoint operation to ensure the capsule can resume.

Recording the checkpoint data is primarily an I/O-bound operation, and by far the most costly portion of the checkpoint is writing the address space contents. Therefore, our implementation allows the address spaces to be written asynchronously. Once all other state is recorded, the address spaces of the member processes are duplicated in a copy-on-write mode, and the capsule can then proceed with normal execution while the system simultaneously records the address space contents.

4.1.3.3 Capsule Relocation

A capsule can be easily relocated by restarting it from its recorded state on a different machine, and we implement this functionality with a new system call (`capsule_restart`). Only the owner of the suspended capsule or the root user executing in the administration capsule can restart the capsule. When a process invokes this operation, a trap is made into the kernel, and the system creates a new capsule in which it duplicates the stored state. New processes are created and initialized with the state recorded previously, and parent-child relationships are re-created. Process address spaces are re-loaded, and IPC is re-established. The file system view is re-created, and open file state is restored. Files with unique mappings from the old view (e.g. /tmp) are copied to an appropriate new destination. Name translation tables are loaded, and interface elements are re-mapped onto the local system resources.

At this point, the capsule state is restored to the conditions at the time of its checkpoint. Before restarting capsule execution, our system places a restart signal at the head of the

signal queue for each process. Thus, when the capsule is resumed, its processes will immediately receive the restart signal, thereby enabling them to perform any desired tasks prior to restarting. This is a standard Unix signal that is ignored by default. Once the restart signal has been delivered, all processes are marked runnable, and execution continues as if their had been no interruption. Any system calls that were executing at the time of the checkpoint will either restart automatically or return an error indicating that they should be retried.

4.2 Experience with Compute Capsules

In this section, we report on our experience using the prototype compute capsule implementation in a test environment containing two types of computer systems: (1) a Sun Ultra 5 workstation with a 270MHz UltraSPARC IIi processor and an 8-bit ATI PGX frame buffer, and (2) a Sun Ultra 60 workstation/server with multiple 300MHz UltraSPARC II processors and a Sun Elite3D graphics accelerator. The machines used in our experiments are standard, off-the-shelf systems running the stock environment for our workgroup cluster. They are binary compatible and have installed our software.

4.2.1 Demonstrations of Capsule Persistence

One of the main goals we wanted to achieve with our implementation was to suspend complete login sessions running various graphical applications, reboot the host machine, and then restart the sessions. With the exception of maintaining open TCP connections, we were able to achieve this goal. For example, we created a capsule with the login session depicted in Figure 4-2 on an Ultra 60. This session is running many graphical applications:

- Netscape web browser
- Terminal windows with command shells
- Quake 3D video game
- Xemacs text editor
- Acrobat reader
- Frame Maker word processor
- Applix Java-based spreadsheet
- MPEG movie player
- xv image viewer/editor
- X utilities (clock, performance meter, solitaire, X eyes, etc.)

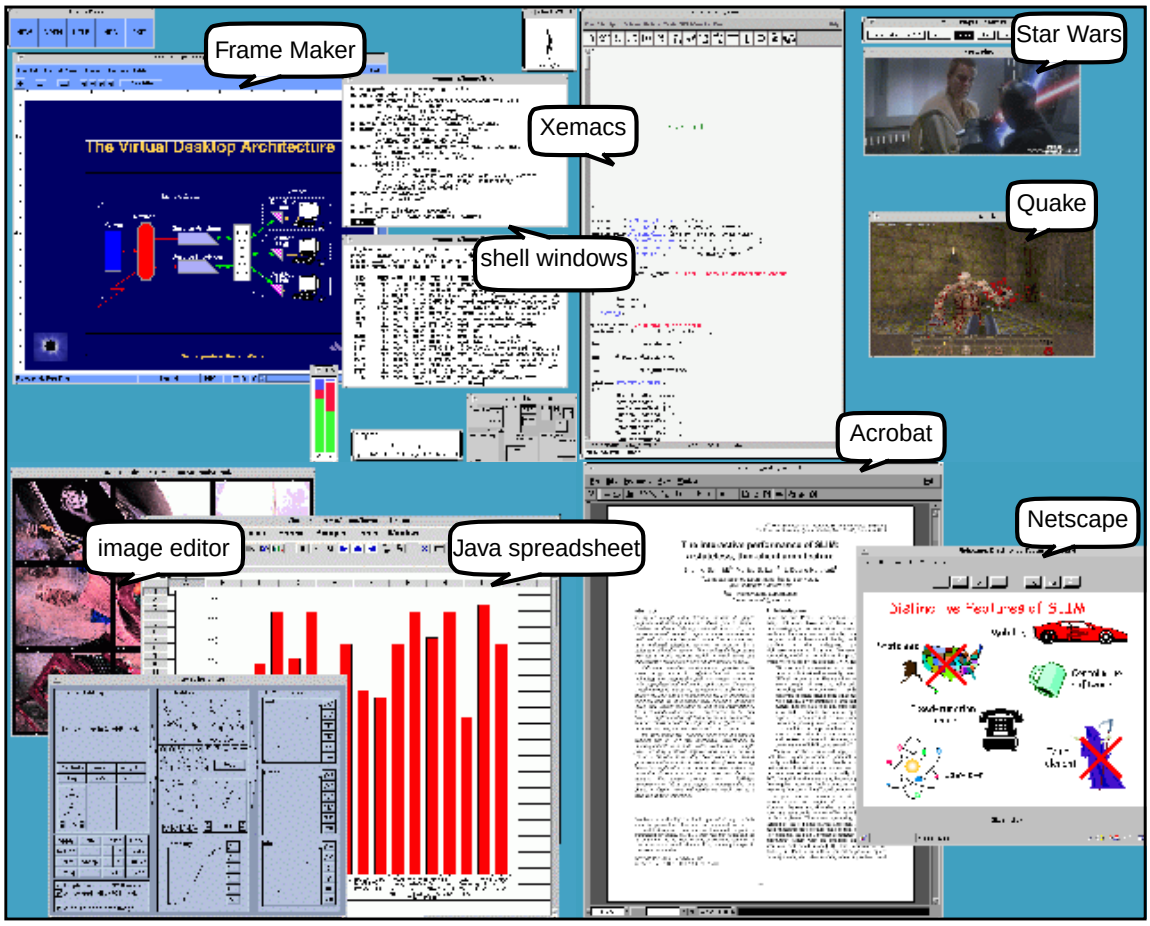


Figure 4-2 Partial screen snapshot of a capsule containing a complete graphical login session.

When the capsule containing this session is suspended and later revived, the processes resume execution where they left off. Quake continues shooting, and the movie continues to play. The Netscape windows display the current web pages (and Java applets), and we can move back through the history of previously visited sites, as well as connect to new locations. The edit buffers are intact for Frame Maker, Xemacs, and xv; and we can undo changes made prior to the suspension. Shell commands (including Unix command pipelines) continue to execute, and we can scroll back through the shell history to view previous commands. The applications have not been modified, communicate in various complex ways, and utilize a wide array of system resources. Still, we can interact with them as if they had never been suspended.

Another ability we wanted to demonstrate was to move an active capsule between binary-compatible machines with different architectures. Because the machines in our system have different graphics controllers, we are restricted to applications that are not display-oriented. Thus, we placed a long-running Verilog simulation with file-based output into a capsule, along with a daemon to checkpoint it periodically. We started the capsule on an Ultra 5, allowed it to make a few checkpoints, and then terminated it ungracefully. On an Ultra 60 machine, we revived the capsule from its last checkpoint, and the Verilog simulation completed its execution, generating the correct final result.

To our knowledge, this is the first demonstration of this level of functionality with respect to suspending active processes and resuming them on a different machine. Fully general-purpose, unmodified, and communicating graphical applications are supported, which has the opportunity to dramatically impact the traditional programming model. When applications are designed with persistence and migration in mind from the outset, we will begin to see new and interesting program behaviors, and it will expose new directions for investigation. For example, a long-hibernating capsule may be revived to find that all the application library resources it was using are no longer available. Closer coordination with the file system is required in this new computing environment.

In addition, assumptions made concerning the state of the world may no longer hold true for older applications. For example, one of the early programs we tried to suspend in a capsule was the Unix ‘man’ command for reading on-line manuals. This program scrolls text in a terminal window a page at a time and then waits for user input before displaying the next page. When this program is suspended while waiting for input, it exits immediately upon revival. The reason is that the command to read input returns with an error indicating that it was interrupted and should be retried. However, the application made the assumption that no such situation should ever arise. While we may argue that the application was poorly designed from the beginning, it serves to point out that such design decisions may be an artifact of a computing model that does not have persistence in mind.

Although we have made substantial progress with our compute capsule implementation, it is a prototype system and therefore still has some limitations. It is not entirely robust and

exhibits some bugs. For example, some applications may sometimes be suspended with incomplete information. While they will still execute correctly once they are revived, we cannot successfully suspend and revive them again. In addition, applications that communicate via Unix System V shared memory are not supported. Fortunately, such applications rarely require such functionality and can operate normally without it. We disabled shared memory on our test systems and were able to run, suspend, and resume the full range of applications we tested, including the X-server, which is the most common application that utilizes shared memory.

Of greater significance is the inability of our system to transparently relocate open network connections. Internet sockets are terminated upon revival, and applications receive a "connection closed" error. Although this restricts the class of applications that can be suspended, it is no different than what is available today with mobile computing devices. When a laptop computer is suspended, it loses all its network connections. Still, some applications are robust enough to handle the loss of open connections and re-establish them as necessary. For example, Netscape will lose any HTTP connections it has open, but it will not crash and can automatically reload the requested pages.

4.2.2 Capsule Overhead

To evaluate the runtime costs associated with namespace translation and maintaining capsules within the system, we added a series of lightweight trace points to our kernel to record the times and durations of various capsule-related activities. We measured our implementation on a Sun Ultra 60 workstation with a 300MHz UltraSPARC II CPU, and our results are summarized in Table 4-2. Note that we have performed no optimizations, and the kernel runs in a debug and event tracing mode.

First, using the capsule creation utility to instantiate a new capsule with a process that immediately exits, we measured the time the system requires to add and remove capsules. The average creation and removal costs were just 74.6µs and 3.8µs respectively. Adding a process to an existing capsule incurs 13.5µs of overhead, and removing a process requires 17.1µs. In contrast, forking a child process and waiting for its (immediate) exit to complete takes 11.3ms, as measured by a user-level program.

Capsule management	Creation	74.6 µs
	Removal	3.8 µs
Process management	Add process to capsule	13.5 µs
	Remove process from capsule	17.1 µs
	Fork and wait for exit	11.3 ms
Virtual interface	Translation	7.0 µs

Table 4-2 Measured capsule overhead on a Sun Ultra60, 300MHz UltraSPARC II CPU.

We measured the cost for each virtual namespace translation to be an average of 7.0µs. To calculate how much overall cost capsules introduced to a running system, we operated the system for a period of 20 minutes, during which we recorded all capsule-related activity. The system was kept busy with a variety of tasks: web browsing, email, video playback, Quake 3D game, various shell commands, etc. Over the test interval, the time spent on capsule-related activities totaled 294ms, or roughly 0.02% overhead.

4.2.3 Checkpoint and Restart Costs

To characterize the costs for capsule persistence and relocation, we checkpointed and restored several capsules encompassing login sessions that varied in number of member processes, open files, IPC channels, pseudo terminals and the total virtual memory size. Based on these experiments, we obtained the formulas listed in Table 4-3 that characterize the time to checkpoint and restore a capsule, as well as the expected size of the state storage. Recording the writable pages of the virtual memory is the dominant cost, but it can be reduced with simple compression techniques. For example, in our experiments an LZW algorithm achieved an average compression ratio of 22:1.

To obtain a better sense of how these costs relate to real-world usage, we took a snapshot of 85 active user sessions on our workgroup server. By recording all the parameters required by our formulas, we calculated the expected costs for each session. On average, checkpoint required 3.05sec (7.58sec maximum), and restart required 3.38sec (7.87sec maximum). The average state size was 75MB, with a maximum of 185MB.

Checkpoint Time	6.347ms/proc + 0.089ms/file + 0.109ms/fifo + 0.085ms/tty + 40.961ms/MB
Restart Time	4.709ms/proc + 0.296ms/file + 0.271ms/fifo + 0.901ms/tty + 40.961ms/MB
State Storage Size	1641B + 4389.2B/proc + 216.4B/file + 357.5B/fifo + 175.0B/tty + VMSIZE

Table 4-3 Costs for checkpointing and restarting a capsule.
The costs are expressed in terms of number of processes, open files, IPC channels, pseudo terminals, and size of writable virtual memory.

4.2.4 Supporting a Campus Computer Lab

To obtain a sense of how capsules might be used in a large-scale deployment, we profiled over 1800 user login sessions at Stanford's student computer center, which consists of 83 workstations available for use by roughly 9000 undergraduate and graduate students. The results are summarized in Table 4-4. Most of the sessions induce very little load (average of 2.35% utilization with occasional bursts of high activity) on the system. Separate research computers are available for handling more highly computation-intensive applications. Since the total system has excess computational bandwidth and each machine is powerful enough to support a large number of users, it is not necessary to actively balance the computational load across the machines in this environment.

Average sizes (per session)	Processes	21
	Read/Write virtual memory	44 MB
	Processor load	2.35%
Average times (per session)	Login duration	1:43 hrs
	Interval between logins	5:11 hrs

Table 4-4 Characteristics of user sessions from 1860 student logins in a university computer lab.

Of concern, however, is the large number of persistent compute capsules that accrue over time. Users may create one or more computing environments that become idle (or forgotten

altogether). These idle capsules take up precious resources, such as process slots, kernel memory, and swap space. In particular, with the average student session requiring 44MB of swap space for writable virtual memory, idle sessions can rapidly consume the 1GB of swap storage available on these machines. Because swap space is allocated from fixed-size, local disks, no increase in swap space can solve this problem completely. Modeling machines as computation caches, we advocate that long-idle capsules be removed and archived in the file system, i.e. placed in expandable storage located on remote, globally accessible file servers. For example, since the average interval of active use is over five hours, capsules containing sessions that have been idle for 30 minutes could be archived. If resources become scarce, such capsules can be safely removed from the system to make more resources available.

4.3 Summary

We have implemented the full range of capsule functionality within the context of a commercial COTS operating system. We added new system calls to manage capsule membership and access control; and the private, modifiable namespace is created through the use of name translation tables inside the kernel and the standard loopback file system. Finally, our implementation includes kernel interfaces between operating system modules for importing and exporting critical state belonging to the members of a capsule. This enables the complete state of a capsule to be extracted from the system, placed in secondary storage, and reconstituted later (possibly on a different machine). Capsule members are free to communicate through standard IPC channels and standard devices, and the implementation can obtain all the necessary state, which has been problematic for systems developed in the past.

The only significant limitation of the prototype implementation is that open network connections between capsules are not transparently moved with the capsule. This is no different than the behavior of laptop computers today when they are disconnected from the network and re-attached in a new location. Although the design presented in Section 2.3.4 could be implemented in a fairly straightforward manner, it would require a burdensome engineering effort and has been left for future work on the system.

Compute capsules are useful for a wide range of purposes. They can be used to enhance system maintenance and extensibility by handing off capsules before scheduled shutdowns or by moving capsules onto newly available machines. They can be used to improve performance by employing modern scheduling techniques to assign a guaranteed level of resources and thereby assure performance isolation, and capsules can be migrated to actively balance resource load. Capsules can also be migrated to support user mobility, or they can be placed in stable storage to enhance system fault tolerance (i.e. a checkpoint and restore facility) or to free scarce resources.

Central to these usage scenarios is the ability to freeze the current state of an active computing session within a capsule and later restore it to the running state, possibly in a different location. We have demonstrated this ability with complete user login sessions containing a large variety of unmodified, real-world applications that interact with each other in an unrestricted manner, something which has not been accomplished before. In addition, the run-time overheads associated with using a capsule system are extremely negligible, and the checkpoint and restart costs are modest and primarily dependent upon disk performance. Finally, we have considered how capsules could be used in a real-world setting to free scarce resources that may be held by long-idle sessions.

5 Related Work

There is no previous system that provides all the features and capabilities that our proposed architecture can support. However, there is a great deal of prior work that is related to the notions of (1) repartitioning systems to remove computing resources from the desktop and (2) moving active computations between machines. In this chapter, we discuss representative examples of the most relevant techniques and contrast them with our approach.

5.1 Thin-Client Systems

Our primary contribution for the decoupling of the human interface from computing resources in the SLIM architecture is the experimental methodology and evaluation of interactive performance, and we discussed related work in Chapter 3. However, with respect to system design, it is useful to compare SLIM to other related systems. There are a wide range of lightweight computing devices that have been developed, including network computers, JavaStations, the Plan 9 Gnot, and network PCs. These machines all differ from the SLIM desktop unit in a significant manner. Although they are low-resource implementations, they are full-fledged computers executing an operating system and windowing software.

Network-transparent access to remote displays was first utilized in the early 1980s by window systems such as VGTS [31] and W [4]. These systems were not portable across operating systems or hardware and relied on synchronous communication primitives, which offered poor performance in the network environments of the day. To address these concerns, the X Window System was developed with a device-independent protocol and fast, asynchronous communication [50], and its design has remained largely intact over the

years. The core concepts of X form the basis of modern thin-client architectures, and we contrast SLIM with such remote display systems in this section.

5.1.1 X Terminals and the X Window System

Because we use X-based applications in our studies, we are able to make the most direct comparison with X Terminals and the X window system. The X protocol [39] was designed to be fairly high-level, with the goal of minimizing bandwidth by expending computing resources locally. SLIM, on the other hand, is designed to minimize local processing, at the cost of a potentially higher bandwidth overhead. Our results, however, show that both approaches are highly competitive. That is, while the SLIM protocol is much simpler and can be serviced by a low-cost processor, it requires roughly the same bandwidth as X for most applications. For higher-bandwidth, image-based applications, SLIM has a significant advantage over X.

In addition, X Terminals are not well-suited to running multimedia applications. Under the X protocol, each frame would have to be transmitted using an `XPutImage` command with no compression possible, i.e. a full 24 bits must be transmitted for each pixel. The situation is different for SLIM. In the worst case, it is the same as X, but typically, some form of compression is possible. If the SLIM CSCS command is not being used, there is still opportunity for finding redundancy among pixels that can reduce bandwidth somewhat. Using the CSCS command, at most 16 bits are required per pixel (a 33% bandwidth reduction) and compression up to 5 bits per pixel (an 88% bandwidth reduction) is possible by altering the color-space conversion parameters. If video is scaled to a higher resolution, the server must perform the operation prior to sending the image to an X Terminal, whereas the Sun Ray 1 console can do it locally at substantial savings in bandwidth. For example, transmitting a half-size video stream to a SLIM console and scaling it to full-size would require roughly 18Mbps of bandwidth. However, over 105Mbps of bandwidth would be needed under X. Thus, SLIM and X are extremely competitive at low bandwidth, while SLIM provides substantial savings on higher bandwidth operations, where its impact is much more significant on overall performance.

5.1.2 Windows-Based Terminals

Windows-based terminals (WinTerms) employ the Citrix ICA protocol [6] or the Microsoft RDP protocol [13][36] to communicate with a server running Windows NT, Terminal Server Edition. These protocols are quite similar in nature to X but are tied to the Windows GUI API. On the other hand, the low-level SLIM protocol has no such bias and can be used by a system with any rendering API. Another difference between the protocols is that ICA and RDP are highly optimized for low-bandwidth connections. This is accomplished via a variety of techniques, including Windows object-specific compression strategies (e.g. run-length coding of bitmaps), caching of Windows objects and state at the terminal, and maintaining backing store at the terminal. Because the resources included in the terminals directly determine the performance and bandwidth savings possible, these types of systems can have expensive terminals that constantly require upgrades to improve performance. They are not fixed-function devices, and they execute a complete operating system. In addition, multimedia capabilities are currently limited, but extra support could be added.

5.1.3 VNC

Like SLIM, the Virtual Network Computing [46][47] (VNC) architecture from Olivetti Research uses a protocol that is independent of any operating system, windowing system, and application. The protocol commands are similar to SLIM, but VNC is designed to access the user's desktop environment from any network connection through a variety of viewers, including a web browser over the internet. The Sun Ray 1 implementation, however, is limited to operating in a workgroup setting with a direct connection to the server.

The key difference between the two approaches, though, is the manner in which the display is updated. With the Sun Ray 1, updates are transmitted from the server to the consoles as they occur in response to application activity. VNC, on the other hand, uses a client-demand approach. Depending on available bandwidth, the VNC viewer periodically requests the current state of the frame buffer. The server responds by transmitting all the pixels that have changed since the last request. This helps the system scale to various bandwidth levels, but has the drawback of larger demands on the server in the form of either

maintaining complex state or calculating a large delta between frame buffer states. In either case, our experience with the system is that even in low-latency, high-bandwidth environments, VNC is fairly sluggish. In addition, VNC includes no support for multimedia applications but leaves open the possibility of including specialized compression techniques in the future.

5.1.4 Other Remote-Display Approaches

The MaxStation [35] from MaxSpeed Corporation is a terminal-based system that uses peripheral cards installed in a PC server machine to transmit video directly to attached consoles. Monitors are refreshed over a 64Mbps connection, but the bandwidth limit restricts the maximum resolution that can be supported to 1024x768 8-bit pixels. The use of expansion cards to create dedicated connections is much less flexible and more difficult to maintain than a commodity switched network, like the Sun Ray 1 uses for the IF.

The Desk Area Network [5][24] (DAN) is a multimedia workstation architecture that uses a high-speed ATM network as the internal interconnect. The frame buffer is a network-attached device that reads and displays pixel tiles. The frame buffer is similar to a Sun Ray 1 desktop unit, but the DAN architecture was designed as a dedicated, stand-alone workstation with sufficient internal bandwidth to transmit high-volume multimedia data (such as video streams) between system components.

5.2 Moving Active Computations

Migrating active computations between machines has been the subject of many research efforts currently and in the past. In this section, we contrast compute capsules with representative examples of the most common approaches.

5.2.1 Virtual Machine Monitors

Virtual machine monitors [22] provide software emulations of physical machines that are identical to the underlying hardware. Examples include Disco [8] and IBM VM/370 [26]. These systems provide several benefits: total isolation between virtual machines, machine independence, migration, and user customization. Because the entire machine is virtualized, the complete operating system serves a purpose similar to that of compute

capsules, i.e. encapsulation of an active computing environment. Because operating systems are self-contained, there are no naming problems (other than for mobile network connections), and there is complete isolation between different instantiations. Virtual machine monitors assign resources to individual operating systems, thereby providing a form of hierarchical resource management. Virtual machine monitors could be used to completely suspend an active instance of an operating system for relocation or off-line storage. Because the operating system itself would be suspended, there is no problem gaining access to hidden state. Each virtual machine monitor has its own disk image, which makes user customization possible. However, because virtual machine monitors merely export a machine hardware interface, they lack the high-level, semantic information available to capsules, and they must encapsulate the entire operating system within a virtual machine.

This forces virtual machines to include unnecessary state. For example, the full range of "physical" memory assigned to the virtual machine must be included, but some is used for caching disk blocks, remote files, and network packets; some is used for kernel structures such as page tables, swap storage tables, device state, and various resource queues; some contains read-only pages of virtual memory; some is used for the kernel itself; etc. Virtual machine monitors must include the entire disk image, but only a small number of files and writable pages of virtual memory are actually needed. Capsules have better semantic knowledge and can be more selective. Similarly, whereas virtual machines cannot convey their resource demands to the monitor in a transparent manner, capsules can benefit from standard schedulers.

Virtual machine monitors also provide no isolation between the operating system and the applications. Processes cannot be moved between virtual machines, and the operating system cannot be modified. Capsules offer greater flexibility by managing processes separately from the operating system. In addition, because capsules leverage the underlying system, they incur no added administration cost. Virtual machine monitors, on the other hand, add maintenance overhead for each monitor instantiated within the system. Finally, virtual machine monitors provide no mechanisms for transparently relocating open network connections, and a capsule-like system would need to be layered on top.

5.2.2 Machine Clusters with a Single System Image

Many distributed operating systems have been developed that provide a single system image across a cluster of machines, i.e. many machines appear as a single unit with a single operating system. Examples include Sprite [40], V [10], Amoeba [54], Charlotte [3], DEMOS/MP [45], and Solaris MC [29]. With varying levels of completeness, these systems provide facilities for migrating individual processes between physical machines in the cluster. This is accomplished by carefully designing the kernels to provide a global namespace and to support location-transparent execution. When a process is migrated, the memory image is moved to the target system, and a variety of optimization techniques are employed. Other process state (such as IPC, device status, and open files) is typically handled by forwarding requests to a home node on which the process originated. Message-based systems, such as V and Charlotte, leave forwarding addresses behind so that communication with system services will remain intact.

A single system image across machines simplifies system management and reduces administration cost, but these types of systems do not scale beyond the local area network environment. Although individual processes can be migrated, groups cannot be treated as a unit for mobility or resource scheduling. Further, there are no mechanisms for suspending processes in off-line storage, and there is no privacy or isolation between users.

In addition to the above systems, GLUnix [21] and Beowulf [48] provide a global namespace on a cluster of machines. Because they merely extend naming support of standard operating systems, these approaches have the benefit of transparently supporting existing applications. These systems, however, merely provide a means for remote execution of processes, which is useful for batch parallel applications but does not meet the needs of a large-scale, remote computational service.

5.2.3 Operating Systems that Support Checkpointing

The Fluke micro-kernel [19] decomposes traditional operating system services into modular and stackable virtual machines that are implemented as nested processes. This nested architecture supports the hierarchical scheduling necessary for making performance guarantees. All kernel objects provide clean mechanisms for importing and exporting their

internal state, which enables processes to easily checkpoint themselves. Because Fluke is a new operating system with a new interface, support for standard computing environments (such as POSIX) is provided by a compatibility layer. Multiple layers can exist on a single machine, which provides privacy and isolation. Unfortunately, these layers are effectively entire UNIX-style operating systems, and they suffer the same inefficiencies as virtual machine monitors, i.e. inclusion of unnecessary state, static assignment of processes to instances of the compatibility layer, and lack of separation between the processes and the compatibility layer. In addition, Fluke is not intended to support distributed collections of machines, does not provide a means for customization, and offers only checkpointing, i.e. active computing environments cannot move to different machines. Further, checkpointing is incomplete, as many objects (such as open files, IPC, and devices) are not included.

KeyKOS [30] and the hypervisor-based system described in [7] are examples of operating systems that record the entire memory image of the kernel and its contents in stable storage. Discount Checking [34] provides similar functionality but logs checkpoint information to reliable memory for better performance. These approaches are very much like the hibernation files recorded by laptop computers when they enter a suspended state. This enables the full system to be restored to its state at the time of the checkpoint, thereby providing a safeguard against failures. Unfortunately, these systems are restricted to restarting on the same machine for failure recovery alone, i.e. they do not provide mobility. In addition, because they merely record the physical memory image of a system, they do not provide any of the other features we require.

5.2.4 User-Level Process Migration

Several systems have been developed to support process migration at the user level outside the operating system. Examples include Condor [33], libckpt [44], CoCheck [55], and MPVM [9]. These systems are primarily intended to support individual, long-running applications on a cluster of machines. An active process can migrate throughout the system, and in some cases a checkpoint facility provides a mechanism for restarting it after a failure. These systems have the benefit of running on top of unmodified operating systems, but they are not suitable for our purposes. Other than migration, they do not provide the

functionality we require, i.e. privacy, isolation, resource management, naming support, user customization, etc. In addition, because there is no kernel support for process migration, these systems offer only incomplete solutions. The state that they can migrate includes the process memory (text, stack, data, heap) and CPU registers, as well as the state of most open files. However, migratory processes are required to be "well-behaved," which means that they cannot use host-dependent services (such as process identifiers), inter-process communication, or special devices. This severely restricts the class of applications that can utilize such systems. Further, processes can be moved individually but not in groups, which duplicates functionality in every application.

5.2.5 Persistent Operating Systems

Operating Systems that support persistence, such as Grasshopper [15], L3 [32], and MONADS [49], combine the functionality of the memory manager and the file system. Objects placed into virtual memory will exist in the persistent store as long as references to them exist. These objects include data structures, files, processes, etc. Although persistent systems enable objects that are normally just cached in memory to have unlimited lifetimes, they do not really provide mechanisms for encapsulating or moving active computations, nor do they provide privacy, isolation, customization, or the other features we demand.

5.2.6 Object-Based Approaches

A variety of systems has been developed to support distributed computations using migrating objects and remote method invocation. Examples include Legion [23], Globe [61], Globus [20], Emerald [28], Rover [27], and Abacus [1]. These systems are implemented as programming languages or middleware toolkits, and objects are programming constructs in the style of object-oriented languages. Some systems support active objects, which include a thread of control. These approaches require explicit programmer control to utilize the mobility features, i.e. checkpoint/restart methods must typically be provided by the programmer. Although this offers greater efficiency in the state that is recorded, it duplicates functionality in every program and becomes unmanageable in complex applications. Worse, it requires all applications to be re-written

in the new programming style, which means there is no support for legacy software. These techniques are too fine-grained and low-level for our purposes.

6 Summary and Conclusions

We are entering a new era of computing in which we will have virtually unlimited resources at our fingertips. Our computing infrastructure is undergoing fundamental changes, shifting away from the traditional, self-contained desktop computing model. The architecture of the future will consist of display consoles in every conceivable location, and they will simply provide access to our active computing environments, which will float around in the network and execute on anonymous collections of resources. This dissertation represents some first steps toward realizing this new model. We have proposed a new system architecture and demonstrated how it can be implemented using stateless display terminals and cacheable computations.

6.1 Decoupling the Human Interface

We have presented the SLIM system, which is a new thin-client architecture. The desktop unit is a stateless, low-cost, fixed-function device that is not much more intelligent than a frame buffer. It requires no system administration and no software or hardware updates. Different applications and operating systems can be ported to display on a SLIM console by simply adding a device driver for the SLIM protocol to their rendering APIs. In addition, the communication requirements are modest and can be supported by commodity interconnection technology.

Our work makes three important contributions. First, we developed an evaluation methodology to characterize interactive performance of modern systems. Second, we characterized the I/O properties of real-world, interactive applications executing in a thin-client environment during normal operation. Finally, via a set of experiments on an actual implementation of the SLIM system, we have shown that it is feasible to create an

implementation that provides excellent quality of service. More specifically, we demonstrated the following:

- Using a dedicated network ensures that added round-trip latency (less than 550μs) between the desktop and the remote server is so low that it cannot be distinguished from sitting directly at the console local to the server.

- By only transmitting encoded pixel updates, network traffic requirements of common, interactive programs are well within the capabilities of modern 100Mbps technology. In fact, 10Mbps is more than adequate, and even 1Mbps provides reasonable performance.

- The low-level SLIM protocol is surprisingly effective, requiring a factor of 2–10 less bandwidth to encode display updates than sending the raw pixel data. Further, its bandwidth demands are not much different from those of the X protocol, i.e. the simplicity of the protocol did not result in the expected increase in bandwidth cost.

- Yardstick applications quantified multi-user interactive performance and showed that substantial amounts of sharing are possible, e.g. the yardstick application and 10 to 36 other active users can share a processor with no noticeable degradation. Network sharing is an order of magnitude larger than processor sharing.

- The consoles are more than adequate to support even high-demand multimedia applications, and server performance turns out to be the main bottleneck. High-resolution streaming video and the Quake video game can be played at rates that offer a high fidelity user experience.

The SLIM architecture provides an effective means of decoupling human interface devices from the servers such that users cannot tell that they are using a remote console. At the same time, users can access their session from any terminal while remaining immune to failures at the desktop, and the consoles require true zero administration.

6.2 Supporting Remote Computational Service

In addition to providing a means for accessing network-resident computing sessions, we must manage the resources on which the computations execute. To support remote computational service, we have argued for the use of mobile capsules of immortal processes to encapsulate active computations that are cached on anonymous hosts in a globally distributed system. This work presents the design, implementation, and evaluation of the new compute capsule abstraction to support this computing model.

Compute capsules are private, portable, persistent computing environments with active processes, and they include the following key features. Capsules contain private namespaces, which provide isolation between capsules as well as host-independent access to the underlying system. They include a modifiable view of a global file system, thereby enabling personalization in a shared setting. Capsules are system management units, i.e. they may be assigned resources and subjected to security constraints or other system policies. Using a new kernel interface to import and export previously inaccessible state, capsules are completely self-contained and may be suspended in stable storage or migrated between machines. Finally, capsules split administration tasks with the host system so that capsule owners have the ability to control and personalize their environments while leveraging professional administration of the underlying system as much as possible.

We have implemented compute capsules within the context of the Solaris 7 Operating Environment for two classes of Sun SPARC machines. The added code is an extremely thin layer of software at the interface between the kernel and applications. The bulk of the implementation exists as a new kernel module, thereby ensuring that no application-level modifications are necessary, and all programs run unchanged.

A major benefit of our capsule implementation is that it offers immortality to groups of active processes. To demonstrate this functionality, we have created, suspended, migrated, and resumed a variety of capsules containing standard applications available today. For example, we created capsules containing complete user-level window sessions with many GUI and multimedia applications, e.g. Netscape, MPEG movie player, Quake 3D game, Frame Maker word processor, etc. We have suspended these capsules in mid operation,

stored them on disk, and restarted them on different machines with their exact state (including graphical I/O and all IPC) at the time of suspension.

We also analyzed the overhead introduced by our implementation. During normal operation, using compute capsules increases the execution cost over a system without capsules by roughly 0.02%. The cost of checkpointing a capsule is relatively modest, requiring about 6.4ms/process and 41ms/MB of writable address space, which can be written asynchronously to allow execution to proceed in milliseconds. Based on our analysis of a large user population, the average time to fully checkpoint and restart a login session is a little over 3sec each.

6.3 Future Directions

Our experience with the new system architecture described in this thesis has exposed many possible directions for future research. One of the most immediate topics to investigate is how to design the distributed service layer. This control layer should include functionality such as a dynamic load balancing system, an automated checkpoint and resume facility to enhance fault tolerance, and automated support for on-line maintenance. In addition, the service layer must coordinate the activity and migration of capsules within a global system. For example, what happens when a capsule must cross an administrative domain? Further, the functionality and capacity of the directory service must be expanded to operate on a truly global scale.

The file system holds many opportunities for innovation. Currently, capsules identify file resources by pathnames. However, it is unreasonable to assume a globally consistent namespace, and file objects may move or change while a capsule is hibernating. Certain file objects, such as application libraries, should be identified by a logical name instead of a path, e.g. DirectX version 6.0 versus C:\WINDOWS\SYSTEM\DRV\DIRECTX6.DLL. In addition, file system activity must be coordinated with capsule status. For example, the file system should not modify or delete files for which a suspended capsule has an external reference. This may require some form of reference counting and garbage collection in the file system. Finally, there are many issues related to caching, replication, and naming in a

truly global file system. The web provides an excellent foundation in this area, but it is a read-only repository.

Perhaps the most obvious direction for future work lies in the area of security. The SLIM protocol operates in the clear and could be easily monitored or spoofed. Encryption can help, but it places substantial processing demands on the client, potentially requiring new hardware and removing the SLIM console from the category of fixed-function appliances. There are also no security mechanisms included with compute capsules. What happens when a user migrates a capsule containing sensitive information from work to home? When users travel and their capsules migrate off their home systems, how can they feel safe to entrust the execution of their capsules to foreign entities? Clearly, much work must be done in this area.

Capsules also create a new computing model that is worthy of further exploration from the standpoint of system design. When computational entities are free to roam throughout the system, new architectures can be developed to take advantage of such functionality. How do software components interact in this new environment? What programming support is necessary? How will resource management and discovery change to fit this model?

Finally, there are more engineering-oriented avenues to explore. The prototype compute capsule implementation must be made more robust to enable wide-scale deployment and experimentation. In addition, neglected features, such as Unix System V shared memory and network connections that can be relocated, must be added to the system to increase its usefulness. We foresee no problem in accomplishing these implementation tasks. However, they must be completed before further experimentation with the system can be carried out, and this is part of our ongoing efforts.

6.4 Conclusions

We have demonstrated the feasibility of the SLIM architecture and compute capsules to provide the necessary support to realize the system architecture of the future. SLIM consoles provide access to remote computations with an interactive experience equivalent to using the local console of a server. Compute capsules represent active computations that

may be cached on any machine of the appropriate architecture. This enables computing sessions to float around on a sea of anonymous and unlimited resources. In addition, they can be used to enhance the scalability, maintainability, reliability, and performance of the overall system. We have combined dedicated, personal computing with time-sharing systems. Users enjoy a high-quality, interactive experience in a shared system that can give a guarantee of performance, permit personal customization, and support persistence of their active computing environments. This work has described the mechanisms necessary to support remote access to persistent computations that execute on a globally distributed set of anonymous computing resources, and we believe it forms the foundation of our future computing infrastructure.

Bibliography

1. K. Amiri, D. Petrou, G. Ganger, and G. Gibson, "Dynamic Function Placement in Active Storage Clusters," *Technical Report CMU-CS-99-140*, School of Computer Science, Carnegie Mellon University, June 1999.

2. T. Anderson, D. Culler, and D. Patterson, "A Case for Networks of Workstations: NOW," *IEEE Micro*, 15(1), February 1995, pp. 54–64.

3. Y. Artsy and R. Finkel, "Designing a Process Migration Facility: The Charlotte Experience," *IEEE Computer*, 22(9), Sept. 1989, pp. 47–56.

4. P. Asente, "W Reference Manual," Internal Document, Computer Science Department, Stanford University, 1984.

5. P. Barham, M. Hayter, D. MacAuley, and I. Pratt, "Devices on the Desk Area Network," *IEEE Journal on Selected Areas in Communications*, 13(4), May 1995, pp. 722–32.

6. Boca Research, Inc., "Citrix ICA Technology Brief," *Technical White Paper*, Boca Raton, FL, 1999.

7. T. Bressoud and F. Schneider, "Hypervisor-Based Fault Tolerance," *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, Dec. 1995, pp. 1–11.

8. E. Bugnion, S. Devine, and M. Rosenblum, "Disco: Running Commodity Operating Systems on Scalable Multiprocessors," *ACM Transactions on Computer Systems*, 15(4), Nov. 1997, pp. 412–447.

9. J. Casas, D. Clark, R. Konuru, S. Otto, R. Prouty, and J. Walpole, "MPVM A Migration Transparent version of PVM," *Computing Systems*, 8(2), Spring 1995, pp. 171–216.

10. D. Cheriton, "The V Distributed System," *Communications of the ACM*, 31(3), March 1988, pp. 314–333.

11. Compaq Computer Corporation, "Performance and Sizing of Compaq Servers with Microsoft Windows NT Server 4.0, Terminal Server Edition," *Technology Brief*, Houston, TX, June 1998.

12. Compaq Computer Corporation, "WinFrame — Scalability and Application Guide," *Technical White Paper*, Houston, TX, January 1997.

13. Databeam Corporation, "A Primer on the T.120 Series Standard," *Technical White Paper*, Lexington, KY, May 1997.

14. A. Dearle, "Toward Ubiquitous Environments for Mobile Users," *IEEE Internet Computing*, January-February 1998, pp. 22–32.

15. A. Dearle, R. di Bona, J. Farrow, F. Henskens, A. Lindstrom, and J. Rosenberg, "Grasshopper, An Orthogonally Persistent Operating System," *Computing Systems*, 7(3), Summer 1994, pp. 289–312.

16. R. Droms, "Dynamic Host Configuration Protocol," *IETF RFC 2131*, March 1997.

17. K. Egevang and P. Francis, "The IP Network Address Translator (NAT)," *IETF RFC 1631*, May 1994.

18. Y. Endo, Z. Wang, J. Chen, and M. Seltzer, "Using Latency to Evaluate Interactive System Performance," *Symposium on Operating System Design and Implementation*, October 1996, pp. 185–199.

19. B. Ford, M. Hibler, J. Lepreau, P. Tullmann, G. Back, and S. Clawson, "Microkernels Meet Recursive Virtual Machines," *Proceedings of the Symposium on Operating System Design and Implementation*, Oct. 1996, pp. 137–151.

20. I. Foster and C. Kesselman, "Globus: A Metacomputing Infrastructure Toolkit," *International Journal of Supercomputer Applications*, 11(2), Summer 1997, pp. 115–128.

21. D. Ghormley, D. Petrou, S. Rodrigues, A. Vahdat, and T. Anderson, "GLUnix: A Global Layer Unix for a Network of Workstations," *Software – Practice and Experience*, 28(9), July 1998, pp. 929–961.

22. R. Goldberg, "Survey of Virtual Machine Research," *IEEE Computer*, 7(6), June 1974, pp. 34–45.

23. A. Grimshaw, A. Ferrari, F. Knabe, M. Humphrey, "Legion: An Operating System for Wide-Area Computing," *Technical Report CS-99-12*, Dept. of Computer Science, University of Virginia, Mar. 1999.

24. M. Hayter and D. McAuley, "The Desk Area Network," *Operating Systems Review*, 25(4), October 1991, pp. 14–21.

25. J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West, "Scale and Performance in a Distributed File System," *ACM Transactions on Computer Systems*, 6(1), Feb. 1988, pp. 51–81.

26. IBM Corporation, *IBM Virtual Machine/370 Planning Guide*, 1972.

27. A. Joseph, J. Tauber, and M. Kaashoek, "Mobile Computing with the Rover Toolkit," *IEEE Transactions on Computers*, 46(3), Mar. 1997, pp. 337–352.

28. E. Jul, H. Levy, N. Hutchinson, and A. Black, "Fine-Grained Mobility in the Emerald System," *ACM Transactions on Computer Systems*, 6(1), Feb. 1988, pp. 109–133.

29. Y. Khalidi, J. Bernabeu, V. Matena, K. Shirriff, and M. Thadani, "Solaris MC: A Multi-Computer OS," *Proceedings of the USENIX 1996 Annual Technical Conference*, Jan. 1996, pp. 191–203.

30. C. Landau, "The Checkpoint Mechanism in KeyKOS," *Proceedings of the 2nd International Workshop on Object Orientation in Operating Systems*, Sept. 1992, pp. 86–91.

31. K. Lantz and W. Nowicki, "Structured Graphics for Distributed Systems," *ACM Transactions on Graphics*, 3(1), January 1984, pp. 23–51.

32. J. Liedtke, "A Persistent System in Real Use — Experiences of the First 13 Years," *Proceedings of the 3rd International Workshop on Object Orientation in Operating Systems*, Dec. 1993, pp. 2–11.

33. M. Litzkow, M. Livny, and M. Mutka, "Condor — A Hunter of Idle Workstations," *Proceedings of the 8th International Conference on Distributed Computing Systems*, June 1988, pp. 104–111.

34. D. Lowell and P. Chen, "Discount Checking: Transparent, Low-Overhead Recovery for General Applications," *Technical Report CSE-TR-410-99*, Dept. of Electrical Engineering and Computer Science, University of Michigan, Nov. 1998.

35. Maxspeed Corporation, "Ultra-Thin Client — Client/Server Architecture: MaxStations under Multiuser Windows 95," *Technical White Paper*, Palo Alto, CA, 1996.

36. Microsoft Corporation, "Comparing MS Windows NT Server 4.0, Terminal Server Edition, and UNIX Application Deployment Solutions," *Technical White Paper*, Redmond, WA, 1999.

37. J. Nieh and M. Lam, "The Design, Implementation and Evaluation of SMART: A Scheduler for Multimedia Applications," *Operating Systems Review*, 31(5), December 1997, pp. 184–97.

38. J. D. Northcutt, J. Hanko, G. Wall, A. Ruberg, "Towards a Virtual Display Architecture," *Project Technical Report SMLI 98-0184*, Sun Microsystems Laboratories, 1998.

39. A. Nye (Ed.), *X Protocol Reference Manual*, O'Reilly & Associates, Inc., 1992.

40. J. Ousterhout, A. Cherenson, F. Douglis, M. Nelson and B. Welch, "The Sprite Network Operating System," *IEEE Computer*, 21(2), Feb. 1988, pp. 23–36.

41. C. Perkins (Ed.), "IP Mobility Support," *IETF RFC 2002*, October 1996.

42. R. Pike, D. Presotto, K. Thompson, and H. Trickey, "Plan 9 from Bell Labs," *Proceedings of the Summer 1990 UKUUG Conference*, July 1990, pp. 1–9.

43. R. Pike, D. Presotto, K. Thompson, H. Trickey, and P. Winterbottom, "The Use of Name Spaces in Plan 9," *Operating Systems Review*, 27(2), Apr. 1993, pp. 72–76.

44. J. Plank, M. Beck, G. Kingsley, and K. Li, "Libckpt: Transparent Checkpointing under Unix," *Proceedings of the USENIX Winter 1995 Technical Conference*, January 1995, pp. 213–224.

45. M. Powell and B. Miller, "Process Migration in DEMOS/MP," *Proceedings of the 9th ACM Symposium on Operating Systems Principles*, Oct. 1983, pp. 110–119.

46. T. Richardson, F. Bennet, G. Mapp, and A. Hopper, "Teleporting in an X Window System Environment," *IEEE Personal Communications*, No. 3, 1994, pp. 6–12.

47. T. Richardson, Q. Stafford-Fraser, K. Wood, and A. Hopper, "Virtual Network Computing," *IEEE Internet Computing*, January–February 1998, pp. 33–38.

48. D. Ridge, D. Becker, P. Merkey, T. Sterling, "Beowulf: Harnessing the Power of Parallelism in a Pile-of-PCs," *Proceedings of the IEEE Aerospace Conference*, Feb. 1997, pp. 79–91.

49. J. Rosenberg and D. Abramson, "MONADS-PC: A Capability Workstation to Support Software Engineering," *Proceedings of the 18th Hawaii International Conference on System Sciences*, Jan. 1985, pp. 222–231.

50. R. Scheifler and J. Gettys, "The X Window System," *ACM Transactions on Graphics*, 5(2), April 1986, pp. 79–109.

51. B. Schmidt, M. Lam, and J. D. Northcutt, "The Interactive Performance of SLIM: a Stateless, Thin-Client Architecture," *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, Dec. 1999, pp. 32–47.

52. M. Schneider and L. Butcher, "NewT Human Interface Device Mark II Terminal Hardware Specification," *Project Technical Report SMLI 98-0200*, Sun Microsystems Laboratories, Palo Alto, CA, 1998.

53. B. Shneiderman, *Designing the User Interface: Strategies for Effective Human-Computer Interaction, 2nd edition*, Addison-Wesley, Reading, MA, 1992.

54. C. Steketee, P. Socko, and B. Kiepuszewski, "Experiences with the Implementation of a Process Migration Mechanism for Amoeba," *Proceedings of the 19th Australasian Computer Science Conference*, Jan. 1996, pp. 140–148.

55. G. Stellner, "CoCheck: Checkpointing and Process Migration for MPI," *Proceedings of the 10th International Parallel Processing Symposium*, April 1996, pp. 526–531.

56. Sun Microsystems Inc., "Java Server Sizing Guide," *Technical White Paper*, Palo Alto, CA, October 1997.

57. Sun Microsystems, Inc., "NFS: Network File System Protocol Specification," *RFC 1094*, Network Working Group, March 1989.

58. A. Tamches and B. Miller, "Fine-Grain Dynamic Instrumentation of Commodity Operating System Kernels," *Symposium on Operating Systems Design and Implementation*, February 1999, pp. 117–30.

59. Unisys, "Sizing and Performance Analysis of Microsoft Windows NT Server 4.0, Terminal Server Edition, on Unisys Aquanta Servers," *Technical White Paper*, Blue Bell, PA, August 1998.

60. A. Vahdat, P. Eastham, C. Yoshikawa, E. Belani, T. Anderson, D. Culler, and M. Dahlin, "WebOS: Operating System Services for Wide Area Applications," *Proceedings of the 7th High Performance Distributed Computing Conference*, July 1998, pp. 52–63.

61. M. Van Steen, P. Homburg, and A. Tanenbaum, "Globe: A Wide-Area Distributed System," *IEEE Concurrency*, 7(1), Jan. 1999, pp. 70–78.